Verfahren zur Verarbeitung von XML-Werten in SQL-Anfrageergebnissen

Dissertation

zur Erlangung des akademischen Grades
Doktor-Ingenieur (Dr.-Ing.)

vorgelegt dem Rat der Fakultät für Mathematik und Informatik der
Friedrich-Schiller-Universität Jena

von
Dipl.-Inf. Thomas Müller
geboren am 5. Juli 1976 in Sondershausen

Gutachter:

1. Prof. Dr. Klaus Küspert, Friedrich-Schiller-Universität Jena
2. Prof. Dr. Uta Störl, Hochschule Darmstadt
3. Dr. Harald Schöning, Software AG, Darmstadt

Tag der Einreichung: 02.04.2008

Tag der letzten Prüfung des Rigorosums: 17.07.2008

Tag der öffentlichen Verteidigung: 25.07.2008

Meiner Familie

Danksagung

Meinem Doktorvater, Herrn Prof. Dr. Klaus Küspert, möchte ich dafür danken, dass er mich während meines Studiums für die Datenbankthematik begeisterte und dass er mir nach Abschluss des Studiums die Möglichkeit eröffnete, als Mitarbeiter an seinem Lehrstuhl zu lehren, zu forschen und zu promovieren. Im Besonderen gilt ihm mein Dank für die kontinuierliche fachliche und außerfachliche Unterstützung während meiner Zeit an der Friedrich-Schiller-Universität Jena.

Frau Prof. Dr. Uta Störl von der Hochschule Darmstadt und Herrn Dr. Harald Schöning von der Software AG in Darmstadt danke ich für die Übernahme der beiden weiteren Dissertationsgutachten.

Mein Dank gebührt auch den zahlreichen Studenten, die durch ihre Studien- und Diplomarbeiten wesentlich zum Gelingen der Arbeit beigetragen haben. Hier seien besonders Robert Böhme, Matthias Brosemann, Martin Jatho, Christoph Kiewitt, Roccy Marhold, Sebastian Ott, Gennadi Rabinovitch und Qiguang Yan hervorgehoben.

Carola Eichner sowie meinen (ehemaligen) Kollegen Dr. Klaus Friedel, Dr. Christoph Gollmick, Matthias Liebisch, Dr. Jens Lufter, Peter Pistor, Gennadi Rabinovitch, Dr. Steffen Skatulla, Dr. Knut Stolze und David Wiese danke ich für das angenehme Arbeitsumfeld am Lehrstuhl.

Schließlich möchte ich mich ganz herzlich bei meiner Frau Mandy, meinem Sohn Moritz Lukas sowie meinen Eltern Eva-Maria und Hans-Joachim für die motivierende Unterstützung in den vergangenen Jahren bedanken.

Jena, im Juli 2008 Thomas Müller

Kurzfassung

In der Praxis besteht verstärkt der Wunsch, traditionelle SQL-Daten und XML-Daten *gemeinsam* zu verwalten und *integriert* auszuwerten. Die sich daraus ergebende Notwendigkeit eines „Brückenschlags“ zwischen SQL und XML wurde sowohl von den SQL-Normungsgremien als auch von den Herstellern relationaler Datenbankmanagementsysteme (RDBMS) erkannt: SQL-Norm und RDBMS-Produkte wurden in den letzten Jahren um XML-Funktionalität erweitert.

Gemäß der aktuellen Version der SQL-Norm kann ein Anfrageergebnis beliebige XQuery-Sequenzen als Spaltenwerte enthalten. Dies schließt insbesondere auch aus *mehreren* Sequenzeinträgen bestehende XQuery-Sequenzen ein, deren Knoten mit Typinformationen angereichert sind. Auf die Frage, wie sich derartige (in einem SQL-Anfrageergebnis enthaltene) XQuery-Sequenzen adäquat mit Hilfe eines Anwendungsprogramms verarbeiten lassen, liefert bisher allerdings weder die SQL-Norm noch ein RDBMS-Produkt eine zufriedenstellende Antwort. Zum Schließen dieser Lücke wird in der vorliegenden Arbeit das Verfahren der *Sequenzcursor-basierten Verarbeitung* eingeführt.

Die Grundidee dieses Verfahrens (welches eine angemessene und komfortable Verarbeitung der in einem SQL-Anfrageergebnis enthaltenen XQuery-Sequenzen ermöglicht) besteht darin, mit Hilfe einer neuen Cursorart, den so genannten Sequenzcursorn, in die XQuery-Sequenzen des aktuellen Ergebnistupels einzutauchen, um Sequenzausschnitte zu definieren. Diese Sequenzausschnitte werden dann ins Anwendungsprogramm übertragen und dort lokal verarbeitet. Sofern dabei Änderungen an den Sequenzausschnitten vorgenommen werden, kann das Anwendungsprogramm entscheiden, ob diese Änderungen in die Datenbank (wieder)eingebracht werden sollen.

Als Grundlage für das Definieren und lokale Verarbeiten der Sequenzausschnitte dient eine neuartige Variante, XQuery-Sequenzen zu repräsentieren. Diese im Rahmen der vorliegenden Arbeit eingeführte und ausführlich beschriebene Repräsentationsform zeichnet sich u. a. dadurch aus, dass die im getypten Wert eines Element- oder Attributknotens enthaltenen atomaren Werte (unter gewissen Voraussetzungen) als eigenständige Knoten dargestellt werden.

In der Arbeit wird der Ablauf der Sequenzcursor-basierten Verarbeitung detailliert vorgestellt. Die einzelnen Arbeitsschritte werden dabei anhand eines realitätsnahen Anwendungsszenarios erläutert. Außerdem wird ein Vorschlag zur Erweiterung von SQL erarbeitet. Durch die angestrebte Spracherweiterung soll eine SQL-seitige Unterstützung der Sequenzcursor-basierten Verarbeitung erreicht werden.

Um die praktische Realisierbarkeit des Sequenzcursor-basierten Verarbeitungsablaufs nachzuweisen, wurden die wichtigsten Konzepte prototypisch umgesetzt. Die Architektur und die Funktionsweise des implementierten Prototyps werden in der Arbeit ausführlich beschrieben.

Desweiteren geht die vorliegende Arbeit auf die unterschiedliche XML-Unterstützung durch SQL/XML:2003 und SQL/XML:2006 ein und gibt einen Überblick über die XML-Funktionalität von Oracle 11g Release 1, MS SQL Server 2005 sowie IBM DB2 Universal Database 9.5.

Inhaltsverzeichnis

Abbildungsverzeichnis

Kapitel 1

Einleitung

Im aktuellen Kapitel gehen wir auf die Motivation, die Ziele und die Gliederung der vorliegenden Arbeit ein.

1.1 Motivation

In nahezu allen größeren Anwendungsszenarien werden Daten mit Hilfe von Datenbanksystemen verwaltet. Die wichtigste Rolle spielen dabei Datenbanksysteme, die auf dem relationalen Datenmodell beruhen und SQL (Structured Query Language) als Datenbanksprache verwenden. Diese relationalen Datenbanksysteme waren ursprünglich vor allem darauf ausgelegt, *(fest)strukturierte* Daten zu handhaben.

Ende der 1990er Jahre setzte im IT-Bereich eine Entwicklung ein, die sich stark auf *semistrukturierte* Daten konzentrierte: Im Februar 1998 verabschiedete das W3C (World Wide Web Consortium) die Metaauszeichnungssprache XML (eXtensible Markup Language) als offizielle Empfehlung. Es folgte eine Reihe weiterer XML-naher Spezifikationen. Seit dieser Zeit haben XML-basierte Techniken enorm an Bedeutung gewonnen. Als Hilfsmittel zur Repräsentation, zum Austausch oder zur Verarbeitung von semistrukturierten Daten sind sie aus der heutigen IT-Landschaft nicht mehr wegzudenken.

Mit der zunehmenden Verbreitung von XML wuchs das Bedürfnis, XML-Daten sicher speichern und effizient anfragen zu können. Unweigerlich ergab sich hieraus die Forderung nach der Existenz „XML-fähiger“ Datenbank(management)systeme. Von der Industrie wurden daraufhin zwei Wege eingeschlagen, diese Forderung zu erfüllen. Zum einen wurden so genannte „native“ XML-Datenbankmanagementsysteme entwickelt, die vollständig auf die Speicherung und Verwaltung von XML-Daten ausgerichtet sind. Als prominentester Vertreter sei hierbei der von der Software AG angebotene Tamino XML Server genannt. Zum anderen wurden die — seit vielen Jahren zur Speicherung und Verwaltung großer Datenmengen etablierten, SQL-basierten — relationalen Datenbankmanagementsysteme (RDBMS) um XML-Funktionalität erweitert. Die wichtigsten Beispiele hierfür sind IBM DB2, Oracle sowie der SQL Server von Microsoft.

Bei unseren weiteren Betrachtungen werden wir uns auf den zweiten der beiden Ansätze beschränken. Dieser Ansatz — nämlich die XML-Erweiterung von RDBMS-Produkten — besitzt derzeit zweifelsfrei die höhere Praxisrelevanz. Verantwortlich für den Erfolg dieses,

auf der Nutzung/Beibehaltung/Weiterentwicklung der Datenbanksprache SQL basierenden, Ansatzes sind u. a. die folgenden Gründe:

- SQL ist eine weit verbreitete, etablierte, mächtige, herstellerunabhängige, genormte Datenbanksprache, in deren Nutzung viele Anwendungsprogrammierer versiert sind. Es gibt bisher keine XML-basierte Datenbanksprache, die alle genannten Vorteile aufweist.
- SQL wird von den DBMS-Produkten zahlreicher Hersteller unterstützt. Diese Produkte werden (z. T. bereits seit über zwei Jahrzehnten) stetig weiterentwickelt und kontinuierlich bezüglich Funktionalität, Performance, Skalierbarkeit und Ausfallsicherheit verbessert.
- Die in der Praxis vorgefundene Datenbanklandschaft wird deutlich vom relationalen Datenmodell und den darauf basierenden Datenbanksystemen dominiert. Ein *kompletter* Umstieg auf ein völlig anderes Datenmodell (verbunden mit einem Wechsel der verwendeten Datenbankprodukte) wäre extrem aufwendig.
- Sehr viele Daten liegen bereits relational vor und eine hohe Anzahl von Datenbankanwendungen basiert auf dem Zugriff über SQL. Eine Überführung sämtlicher existierender Daten ins XML-Format und eine Anpassung bzw. Neuentwicklung der Anwendungsprogramme kommen aus Aufwandsgründen i. d. R. nicht in Frage.
- Viele Anwender (wie beispielsweise Unternehmen) sind daran interessiert, XML-Daten und „traditionelle“ SQL-Daten *gemeinsam* verwalten und *integriert* auswerten zu können, ohne dabei auf die bewährten (und zugleich effizienten) SQL-basierten Zugriffsmöglichkeiten verzichten zu müssen.

Als die RDBMS-Hersteller damit begannen, ihre Produkte um XML-Funktionalität zu erweitern, entwickelten sie zunächst proprietäre XML-Erweiterungen der Datenbanksprache SQL. Auf diesen Vorstoß seitens der RDBMS-Anbieter haben die SQL-Normungsgremien dann zeitnah reagiert: Um ein (zu starkes) Auseinanderdriften in unterschiedliche produktspezifische „SQL/XML-Sprachdialekte“ zu verhindern, wurde die von SQL bereitzustellende XML-Funktionalität genormt — die SQL-Norm wurde also um XML-basierte Konzepte erweitert.

Die derzeit gültige SQL-Norm stellt zur Verwaltung von XML-Werten einen Basisdatentyp namens `XML` bereit, bei dessen Instanzen es sich um beliebige XQuery-Sequenzen gemäß des im Jahr 2007 vom W3C verabschiedeten XQuery-Datenmodells handeln kann. Entsprechend der SQL-Norm ist es somit möglich, Tabellen anzulegen, die in einer oder mehreren Spalten beliebige XQuery-Sequenzen enthalten. Solche (möglicherweise sehr großen, aus vielen Sequenzeinträgen bestehenden) XQuery-Sequenzen können auch innerhalb von SQL-Anfrageergebnissen auftreten.

Es existiert (abgesehen von dem im Rahmen der vorliegenden Arbeit vorgestellten Ansatz) noch kein Verfahren, das eine adäquate Verarbeitung der in einem SQL-Anfrageergebnis enthaltenen XQuery-Sequenzen ermöglicht. In Anfrageergebnissen auftretende Instanzen des von der SQL-Norm bereitgestellten Basisdatentyps `XML` können — im Gegensatz zu den Instanzen sämtlicher anderer SQL-Basisdatentypen — bisher also nicht angemessen verarbeitet werden. Die „Praxistauglichkeit“ des SQL-Basisdatentyps `XML` ist damit erheblich eingeschränkt — insbesondere lassen sich keine Anwendungsszenarien realisieren,

die darauf beruhen, dass in Anfrageergebnissen enthaltene Instanzen dieses Datentyps adäquat von einem Anwendungsprogramm verarbeitet werden.

Um die bezüglich des SQL-Basisdatentyps `XML` bestehenden Einschränkungen zu überwinden, ist es erforderlich, ein Verfahren zu entwickeln, das eine angemessene Verarbeitung der in SQL-Anfrageergebnissen enthaltenen XQuery-Sequenzen gestattet. Die Konzeption bzw. die Beschreibung eines entsprechenden Verfahrens bildet den Schwerpunkt der vorliegenden Arbeit.

1.2 Ziele

Die wichtigsten Ziele der vorliegenden Arbeit können wie folgt zusammengefasst werden:

Beschreibung der von der SQL-Norm vorgesehenen XML-Funktionalität
: Es soll aufgezeigt werden, inwieweit die SQL-Norm XML-basierte Konzepte unterstützt. Von besonderem Interesse ist dabei der von der SQL-Norm bereitgestellte Basisdatentyp `XML`, der auf dem XQuery-Datenmodell beruht und als XML-Werte beliebige XQuery-Sequenzen zulässt.

Beschreibung der von den RDBMS-Produkten bereitgestellten XML-Funktionalität
: Es soll erläutert werden, inwieweit XML-Werte (also XML-Dokumente bzw. XQuery-Sequenzen) von den aktuellen Versionen der wichtigsten relationalen Datenbankprodukte unterstützt werden. Dabei soll u. a. auch darauf eingegangen werden, ob die von den Produkten angebotene XML-Funktionalität konform mit den Vorgaben der SQL-Norm ist.

Entwicklung eines realitätsnahen SQL/XML-Anwendungsszenarios
: Es soll ein realitätsnahes, leicht verständliches Beispielszenario entwickelt und vorgestellt werden, bei dem XQuery-Sequenzen, die in einem SQL-Anfrageergebnis enthalten sind, von einem Anwendungsprogramm verarbeitet werden. Das entsprechende Beispielszenario soll (um eine Beschränkung auf triviale Spezialfälle zu vermeiden) so konzipiert sein, dass die zu verarbeitenden XQuery-Sequenzen i. d. R. aus *mehreren* Sequenzeinträgen bestehen.

Konzeption eines Verfahrens zur adäquaten Verarbeitung von XML-Werten
: Es soll ein Verfahren konzipiert und beschrieben werden, welches es ermöglicht, die in einem SQL-Anfrageergebnis enthaltenen (möglicherweise sehr großen) XQuery-Sequenzen adäquat mit Hilfe eines Anwendungsprogramms zu verarbeiten. Die Konzeption bzw. Beschreibung eines solchen Verfahrens stellt das Hauptziel der vorliegenden Arbeit dar.

Erarbeitung eines Sprachvorschlags zur Erweiterung von SQL
: Es soll ein Sprachvorschlag erarbeitet werden, der beschreibt, wie die Datenbanksprache SQL zu erweitern ist, damit sie das im Rahmen der vorliegenden Arbeit vorgeschlagene (der adäquaten Verarbeitung von XML-Werten dienende) Verfahren unterstützt.

Nachweis der Realisierbarkeit des von uns konzipierten Verfahrens
: Es soll nachgewiesen werden, dass das in der vorliegenden Arbeit beschriebene Ver-

fahren tatsächlich realisierbar ist. Der Nachweis der Realisierbarkeit soll mit Hilfe eines Prototyps erbracht werden.

1.3 Gliederung

Im *Kapitel 2* werden wir grundlegende SQL- bzw. XML-bezogene Konzepte vorstellen, einen Überblick über die XML-Unterstützung durch die SQL-Norm geben, auf die XML-Funktionalität heutiger RDBMS-Produkte eingehen und das (als Bezugspunkt für weitere Erläuterungen dienende) Beispielszenario Kundenkartenverwaltung einführen.

Im *Kapitel 3* zeigen wir zunächst, dass sich die traditionelle Repräsentation von XQuery-Sequenzen nicht eignet, um als Basis der im Kapitel 4 vorgestellten Sequenzcursor-basierten Verarbeitung genutzt zu werden. Anschließend werden wir mit der typed-value-orientierten Repräsentation eine neuartige Möglichkeit einführen, XQuery-Sequenzen zu repräsentieren. Diese von uns konzipierte Repräsentationsvariante dient dann als Grundlage für die Sequenzcursor-basierte Verarbeitung.

Im *Kapitel 4* gehen wir detailliert auf das von uns entwickelte Verfahren der Sequenzcursor-basierten Verarbeitung ein, welches eine adäquate und komfortable Verarbeitung der in einem SQL-Anfrageergebnis enthaltenen XQuery-Sequenzen ermöglicht. Wir werden die einzelnen Verarbeitungsschritte dieses Verfahrens ausführlich erläutern, mögliche Erweiterungen des Verarbeitungsablaufs beschreiben, denkbare alternative Verarbeitungsansätze diskutieren und eine Abgrenzung gegen bereits existierende Ansätze vornehmen.

Das Grundprinzip der Sequenzcursor-basierten Verarbeitung besteht darin, dass mit Hilfe von so genannten Sequenzcursorn Ausschnitte der im aktuellen Ergebnistupel enthaltenen XQuery-Sequenzen definiert werden. Diese Sequenzausschnitte werden dann ins Anwendungsprogramm übertragen und dort lokal verarbeitet. Nähere Details zu Sequenzcursorn finden sich im *Kapitel 5*. Sequenzausschnitte werden anschließend ausführlich im *Kapitel 6* behandelt.

Im *Kapitel 7* wird beschrieben, welche Erweiterungen an der Datenbanksprache SQL vorzunehmen sind, damit eine SQL-seitige Unterstützung der Sequenzcursor-basierten Verarbeitung gewährleistet wird. *Kapitel 8* stellt dann eine prototypische Realisierung der wesentlichen Konzepte der Sequenzcursor-basierten Verarbeitung vor. Im Anschluss daran fasst *Kapitel 9* die vorliegende Arbeit nochmals kompakt zusammen und gibt einen Ausblick auf mögliche weiterführende Arbeiten.

Abschließend noch ein Hinweis zur Aktualität der vorliegenden Arbeit: In der Arbeit enthaltene Aussagen über die *aktuelle* Version der SQL-Norm, die *aktuelle* Version einer W3C-Empfehlung oder die *aktuelle* Version eines Produkts beziehen sich auf den Stand vom *März 2008*.

Kapitel 2

Grundlagen

In diesem Kapitel werden grundlegende Begriffe und Konzepte erläutert, die für das Verständnis der vorliegenden Arbeit hilfreich sind. Nach einer Betrachtung SQL-bezogener Themen in den Abschnitten 2.1 bis 2.5 widmen sich die Abschnitte 2.6 bis 2.10 XML-nahen Konzepten. Um die *gemeinsame* Nutzung von SQL und XML geht es anschließend in den Abschnitten 2.11 und 2.12. Im Abschnitt 2.13 wird dann ein Beispielszenario eingeführt, auf welches im weiteren Verlauf der Arbeit wiederholt Bezug genommen wird.

2.1 SQL

Nachdem E. F. Codd im Jahr 1970 das relationale Datenmodell [Cod70] vorgeschlagen hatte, erfolgte durch IBM in den 1970er Jahren eine der ersten prototypischen Realisierungen eines relationalen DBMS. Zu dem als „System R“ bekannt gewordenen Prototyp gehörte die Abfragesprache *SEQUEL (Structured English Query Language)*, welche später unter dem Namen *SQL (Structured Query Language)* weiterentwickelt wurde [DD97].

Um ein Auseinanderdriften in proprietäre Sprachdialekte zu verhindern bzw. einzudämmen, begann 1982 eine Arbeitsgruppe des *American National Standards Institute (ANSI)* mit der Normung von SQL. Die erste Version der SQL-Norm wurde im Jahr 1986 verabschiedet (SQL-86) und im darauffolgenden Jahr von der *International Organization for Standardization (ISO)* übernommen [ISO87]. Nach kontinuierlichen Weiterentwicklungen von SQL veröffentlichte die ISO in den Jahren 1989, 1992, 1999 und 2003 Folgeversionen der SQL-Norm, die als SQL-89 [ISO89], SQL-92 [ISO92], SQL:1999 [ISO99] bzw. SQL:2003 [ISO03a] bezeichnet werden [MKF+03, OP04]. SQL:2003, die derzeit gültige SQL-Norm, wird voraussichtlich im Laufe des Jahres 2008 durch eine neuere Version ersetzt.

SQL hat sich als die wichtigste Datenbanksprache für relationale Datenbanksysteme etabliert und wird (in leichten Abwandlungen) von den DBMS-Produkten zahlreicher Hersteller unterstützt. Als marktbedeutendste kommerzielle Systeme sind hierbei IBM DB2, Oracle sowie — seit einigen Jahren — der SQL Server von Microsoft zu nennen.

2.2 SQL-basierter Datenbankzugriff aus Anwendungsprogrammen

SQL war ursprünglich als *interaktive* Datenbanksprache konzipiert worden: Nachdem der Nutzer seine SQL-Anfrage mittels Tastatur eingegeben hat, wird ihm das Anfrageergebnis am Bildschirm angezeigt. Diese von den „SQL-Vätern“ primär vorgesehene interaktive Arbeitsweise hat sich in der Praxis jedoch nicht durchgesetzt. Stattdessen erfolgen Datenbankzugriffe i. d. R. aus Anwendungsprogrammen heraus [Neu92, Neu96, KBL05]. Ein Grund hierfür — neben anderen — ist die meist beträchtliche Komplexität vieler SQL-Anweisungen.

Bei der Anwendungsentwicklung wird SQL in Kombination mit einer höheren Programmiersprache, wie beispielsweise C, C++, Ada, COBOL, Fortran, Pascal oder Java eingesetzt. Die Einbettung von SQL in eine derartige Programmiersprache ist nicht trivial, da beide Sprachen unterschiedlichen Sprachparadigmen folgen. Diese „Fehlanpassung“ zwischen deskriptivem multimengenorientierten Datenzugriff in SQL und prozeduraler satzweiser Verarbeitung in Programmiersprachen ist unter der Bezeichnung „Impedance Mismatch“ bekannt geworden. Das zur Lösung dieses Problems genutzte Cursor-Konzept werden wir im Abschnitt 2.4 vorstellen.

Für die Einbettung von SQL in eine höhere Programmiersprache existieren verschiedene Möglichkeiten. Die beiden für die Praxis wichtigsten Varianten, die *Einbettung mit Vorübersetzer* und die *prozedurale Schnittstelle*, werden wir im Folgenden kurz erläutern. Auf andere Ansätze, wie *einfache oder komplexe Programmiersprachenerweiterungen* oder *integrierte Datenbankprogrammiersprachen*, werden wir nicht näher eingehen und verweisen stattdessen auf die einschlägige Literatur, z. B. [Neu92].

- *Einbettung mit Vorübersetzer*

  SQL-Anweisungen werden (mit einer speziellen Markierung versehen) in den Quellcode eingestreut. Ein Vorübersetzer *(precompiler)* generiert daraus Quellcode ohne SQL-Anweisungen, indem er die markierten Anweisungen durch Prozeduraufrufe und andere Konstrukte der verwendeten höheren Programmiersprache *(Wirtssprache)* ersetzt. Aus dem so erhaltenen Quellcode wird dann mit Hilfe des normalen Programmiersprachenübersetzers das ausführbare Programm erzeugt. Zur Laufzeit dienen die vom Vorübersetzer eingefügten Prozeduraufrufe zur Kommunikation mit dem DBMS.

  Typische Vertreter dieser Einbettungsvariante sind *Embedded SQL* [Neu96] für Wirtssprachen wie C, C++, Ada, COBOL, Fortran oder Pascal sowie *SQLJ* [ME00] für Java.

- *Prozedurale Schnittstelle*

  Im Gegensatz zur zuvor betrachteten Einbettungsart wird das Programm *komplett* in einer höheren Programmiersprache geschrieben, sodass kein Vorübersetzer erforderlich ist. Der Aufruf der Datenbankfunktionen erfolgt mit Hilfe externer Prozeduren, die von der prozeduralen Schnittstelle bereitgestellt werden.

  Die wichtigsten prozeduralen Schnittstellen sind *ODBC* [Gei95] bzw. *SQL/CLI* [ISO03d] für Programmiersprachen wie C, C++, Ada, COBOL, Fortran oder Pascal sowie *JDBC* [Ree00, Sun06] für Java.

Wir haben den in dieser Arbeit vorgestellten Sequenzcursor-basierten Verarbeitungsablauf unabhängig von einer konkreten Programmiersprache entworfen. Sofern es im weiteren Verlauf dieser Arbeit jedoch sinnvoll bzw. notwendig ist, auf eine konkrete Programmiersprachensyntax bzw. -funktionalität Bezug zu nehmen, werden wir C++ als verwendete Sprache voraussetzen. Die in Anlehnung an C++ vorgestellten Konzepte und Arbeitsschritte lassen sich (unter Berücksichtigung der syntaktischen und methodischen Unterschiede zwischen den einzelnen Sprachen) prinzipiell auch auf andere höhere Programmiersprachen übertragen.

In der vorliegenden Arbeit werden wir die einzelnen Arbeitsschritte des Sequenzcursor-basierten Verarbeitungsablaufs auf der Grundlage von Embedded SQL beschreiben. Das Konzept der Sequenzcursor-basierten Verarbeitung ist jedoch unabhängig von einer konkreten Einbettungsvariante. Der basierend auf Embedded SQL beschriebene Verarbeitungsablauf lässt sich (unter Beachtung der konzeptionellen Unterschiede zwischen den verschiedenen Einbettungsmöglichkeiten) prinzipiell auch auf andere Einbettungsvarianten übertragen.

2.3 Embedded SQL

Wir wollen im Folgenden kurz auf ein paar Aspekte von Embedded SQL *(ESQL)* eingehen, die für das Verständnis dieser Arbeit hilfreich sind. Für eine umfassende Beschreibung sei auf die entsprechende Fachliteratur verwiesen, z. B. [Neu96].

2.3.1 Statisches vs. dynamisches Embedded SQL

Bei Embedded SQL kann zwischen einer statischen und einer dynamischen Form unterschieden werden. Wir werden beide Möglichkeiten anhand eines kleinen Beispiels gegenüberstellen. Wir setzen dabei eine Tabelle namens *Kunde* voraus, die u. a. die Spalten *KNr* (Kundennummer) und *Ort* (Wohnort) enthält und betrachten eine `UPDATE`-Anweisung, die den Wohnort eines bestimmten Kunden ändert.

- *Statisches Embedded SQL*

  Die SQL-Anweisungen werden (mit dem Präfix `EXEC SQL` markiert) direkt in den Quelltext geschrieben *(Abbildung 2.1)*. Dabei können anstelle von konstanten Werten (in unserem Beispiel *'Jena'* und *4711*) auch Hostvariablen verwendet werden.

  ```
  EXEC SQL UPDATE Kunde SET Ort='Jena' WHERE KNr=4711;
  ```

  Abbildung 2.1: Beispiel für eine `UPDATE`-Anweisung in statischem ESQL

  Da die Anweisungen (bis auf die Belegung der Hostvariablen) bereits zur Vorübersetzungszeit bekannt sein müssen (sie stehen ja „hart-verdrahtet“ im Quellcode), können bei der Vorübersetzung bereits zahlreiche Fehler abgefangen werden. Diesem Vorteil steht als entscheidender Nachteil jedoch die große Inflexibilität gegenüber.

- *Dynamisches Embedded SQL*

  Die SQL-Anweisungen werden zur Laufzeit in Form von Zeichenketten übergeben.

Mit Hilfe einer (statisch eingebetteten) `EXECUTE-IMMEDIATE`-Anweisung kann dann die Anweisungsübersetzung und -ausführung initiiert werden *(Abbildung 2.2)*. Alternativ kann auch eine `PREPARE`-Anweisung, gefolgt von einer `EXECUTE`-Anweisung, genutzt werden, wodurch sich Anweisungsübersetzung und -ausführung zeitlich trennen lassen.

```
strcpy(anw, "UPDATE Kunde SET Ort='Jena' WHERE KNr=4711");
EXEC SQL EXECUTE IMMEDIATE :anw;
```

Abbildung 2.2: Beispiel für eine `UPDATE`-Anweisung in dynamischem ESQL

Da die Anweisungen dynamisch zur Laufzeit zusammengebaut werden können, ist ein sehr hohes Maß an Flexibilität gewährleistet. Andererseits ist es damit natürlich nicht möglich, fehlerhafte SQL-Anweisungen bereits während der Vorübersetzung zu erkennen, was zu einer höheren Fehleranfälligkeit zur Laufzeit führt.

Die SQL-`SELECT`-Anweisung erfordert allerdings eine Sonderbehandlung und lässt sich weder mittels statischen noch mittels dynamischen Embedded SQL wie zuvor beschrieben einbetten. Wir werden hierauf im Abschnitt 2.4 näher eingehen. Lediglich wenn bekannt ist, dass das Anfrageergebnis garantiert höchstens ein Ergebnistupel enthält, kann eine Abwandlung der `SELECT`-Anweisung — die Anweisung `SELECT INTO` — genutzt werden, welche sich statisch in den Quelltext einbetten lässt.

2.3.2 Fehler- und Ausnahmebehandlung

Die Ausführung einer (eingebetteten) SQL-Anweisung kann scheitern. Betrachten wir hierzu beispielsweise unsere Kundentabelle aus dem vorigen Abschnitt und nehmen nun zusätzlich an, die Spalte *KNr* sei als Primärschlüssel definiert worden. Wird nun versucht, einen Kunden in die Tabelle einzufügen, obwohl bereits ein Kunde mit derselben Kundennummer existiert, so scheitert die entsprechende `INSERT`-Anweisung.

Um dem Anwendungsprogramm das Gelingen oder Scheitern der Anfrageausführung mitzuteilen, werden i. d. R. zwei spezielle Hostvariablen namens `SQLSTATE` und `SQLCODE` genutzt [DD97]. Das SQL-Laufzeitsystem aktualisiert die Werte beider Variablen nach jeder Anweisungsausführung. Durch ein Abfragen dieser Werte kann das Anwendungsprogramm feststellen, ob die *zuletzt* ausgeführte Anweisung erfolgreich war. Damit ist es dem Anwendungsprogramm möglich, geeignet auf Ausnahme- bzw. Fehlersituationen zu reagieren.

Bei dem vom SQL-Laufzeitsystem zurückgelieferten `SQLSTATE`-Wert handelt es sich um eine Zeichenkette der Länge 5. Beispielsweise würde bei der von uns angenommenen (drohenden) Verletzung der Primärschlüsseleigenschaft der `SQLSTATE`-Wert auf *'23505'* gesetzt. Die ersten beiden Ziffern kodieren dabei die allgemeine Fehlerklasse *(23 = Integritätsverletzung)*, während die restlichen Ziffern die Fehlerursache weiter eingrenzen *(hier: 505 = unzulässiger Duplikatwert)*. Der `SQLSTATE`-Wert *'00000'* signalisiert eine erfolgreiche Ausführung der SQL-Anweisung.

Während die `SQLSTATE`-Werte (zumindest teilweise) durch SQL:2003 genormt sind, werden die `SQLCODE`-Werte nicht mehr von der SQL-Norm erfasst. Die `SQLCODE`-relevanten Konventionen, die früher Bestandteil der SQL-Norm waren, haben in der Praxis dennoch ihre Gültigkeit behalten. So handelt es sich bei den `SQLCODE`-Werten um Integer-Werte, wobei

negative Zahlen für einen Fehler und positive Zahlen für eine Warnung stehen. Der Wert *0* signalisiert eine erfolgreiche Anweisungsausführung und der Wert *100* bedeutet, dass (beispielsweise bei einer `FETCH`-Anweisung) kein Tupel (mehr) gefunden wurde. Alle anderen `SQLCODE`-Werte sind vom konkreten DBMS-Produkt abhängig. Im von uns unterstellten Szenario würde also ein negativer `SQLCODE`-Wert (z. B. *-803* in DB2) zurückgeliefert.

Die Variablen `SQLCODE` und `SQLSTATE` können auch Bestandteil einer so genannten *SQLCA (SQL Communication Area)* sein [HS00, IBM07b]. Dabei handelt es sich um eine (nicht genormte) Zusammenfassung verschiedener Statusvariablen, welche nach jeder Anweisungsausführung aktualisiert werden. Die SQLCA lässt sich mittels `EXEC SQL INCLUDE SQLCA` ins Anwendungsprogramm einbinden und stellt diesem detaillierte Informationen über aufgetretene Fehler bereit.

2.3.3 Hostvariablen

Variablen, die gemeinsam vom Anwendungsprogramm und vom SQL-Laufzeitsystem genutzt werden, heißen *Hostvariablen* [DD97]. Mit ihrer Hilfe ist es möglich, Werte zwischen dem Anwendungsprogramm und dem SQL-Laufzeitsystem auszutauschen. Beispielsweise dienen die im vorigen Abschnitt betrachteten Hostvariablen *SQLSTATE* und *SQLCODE* zur Übergabe von Statusinformationen an das Anwendungsprogramm. Werden Hostvariablen hingegen in einer SQL-Anweisung anstelle von Konstanten verwendet, erfolgt damit eine Übergabe ihrer aktuellen Wertebelegung an das SQL-Laufzeitsystem.

Hostvariablen werden in einer so genannten *embedded SQL declare section* deklariert. Innerhalb von SQL-Anweisungen werden sie mit einem vorangestellten Doppelpunkt gekennzeichnet, um sie von (möglicherweise gleichnamigen) Spalten unterscheiden zu können.

2.4 SQL-Cursorkonzept

Das Ergebnis einer SQL-`SELECT`-Anweisung kann (beliebig) viele Tupel enthalten. Möglicherweise möchte das Anwendungsprogramm aber nur einen Teil des Anfrageergebnisses verarbeiten oder es benötigt zu jedem Zeitpunkt nur jeweils ein Ergebnistupel. In beiden Fällen (und in etlichen weiteren Situationen) wäre es nicht sinnvoll, die gesamte Ergebnistupel(multi)menge in einem Schritt ins Anwendungsprogramm zu übertragen. Benötigt wird also eine Technik, die es erlaubt, die Ergebnistupel einzeln (d. h. nacheinander) oder in kleineren „Portionen“ ins Anwendungsprogramm zu übergeben. SQL stellt hierfür das so genannte *Cursorkonzept* bereit [DD97, HS00, KBL05]. Damit wird es (zumindest teilweise) möglich, den Impedance Mismatch (Abschnitt 2.2) zu „überwinden“.

Ein *Tupelcursor* kann als eine spezielle Laufvariable zum tupelweisen Abarbeiten des Anfrageergebnisses angesehen werden. Ein Tupelcursor wird hierzu an eine bestimmte `SELECT`-Anfrage gebunden, deren Ausführung durch das „Öffnen“ des Tupelcursors initiiert wird. Nach seinem Öffnen steht der Tupelcursor vor dem ersten Ergebnistupel. Der Tupelcursor wird nun Schritt für Schritt durch das Anfrageergebnis bewegt, d. h., er wird jeweils auf das nächste Ergebnistupel weiterpositioniert. Mit bzw. nach jeder Bewegung des Tupelcursors wird das aktuelle Ergebnistupel (also das Ergebnistupel, auf dem der Tupelcursor gerade steht) ins Anwendungsprogramm übertragen. Wurde das Anfrageergebnis vollständig abgearbeitet (oder soll die Verarbeitung vorzeitig beendet werden), kann der Tupelcursor geschlossen werden.

Als eine Erweiterung des ursprünglichen Cursorkonzepts erlaubt die SQL-Norm auch so genannte „scrollable cursors“. Dies bedeutet, dass ein Tupelcursor auf dem Anfrageergebnis beliebig vor- und zurückbewegt werden kann. Anstatt (wie bisher) nur auf das jeweils nächste Ergebnistupel positioniert werden zu können, kann nun beispielsweise auch gezielt das vorige, das übernächste, das erste oder das letzte Ergebnistupel angesprungen werden. Für weitere Details verweisen wir auf die entsprechende Fachliteratur, z. B. [HS00, KBL05, ISO03c].

Das Cursorkonzept ist nicht nur für lesende Zugriffe, sondern auch für Änderungs- und Löschoperationen relevant. Hierfür stellt die SQL-Norm spezielle Formen der `UPDATE`- und `DELETE`-Anweisung bereit, die ein Ändern bzw. Löschen mit Bezugnahme auf die aktuelle Tupelcursorposition erlauben. Diese Anweisungen werden als *positioniertes UPDATE* bzw. *positioniertes DELETE* bezeichnet. Wir werden hierauf im Abschnitt 2.4.3 näher eingehen.

Das SQL-Cursorkonzept kann sowohl in statischem als auch in dynamischem Embedded SQL genutzt werden. Wir werden beide Möglichkeiten im Folgenden betrachten und dabei wieder unser Beispiel aus Abschnitt 2.3.1 aufgreifen. Wir nehmen diesmal an, es seien alle Kunden aus Jena gesucht und unterstellen ferner, dass die Kundentabelle auch die Spalten *Name* und *GebDatum* (Geburtsdatum) besitzt. Zusätzlich setzen wir voraus, dass für die Spalte *GebDatum* Nullwerte erlaubt sind, während dies für die restlichen Spalten nicht der Fall sei.

2.4.1 Cursorkonzept und statisches Embedded SQL

Mit Hilfe der `DECLARE`-Anweisung wird ein Tupelcursor deklariert und an die gewünschte `SELECT`-Anfrage gebunden *(Abbildung 2.3)*. Anstelle des konstanten Wertes *('Jena')* hätte dabei auch eine Hostvariable genutzt werden können.

```
EXEC SQL DECLARE unserTC CURSOR FOR
  SELECT KNr, Name, GebDatum FROM Kunde WHERE Ort='Jena';
```

Abbildung 2.3: Beispiel für eine Tupelcursordeklaration in statischem ESQL

Da die `SELECT`-Anfrage als Bestandteil der Deklarationsanweisung „fest-verdrahtet“ im Quelltext steht, muss sie (bis auf die Belegung etwaiger Hostvariablen) bereits zur Vorübersetzungszeit vollständig bekannt sein. Dies hat u. a. zur Folge, dass auch die Struktur des Anfrageergebnisses (Anzahl, Namen und Datentypen der Ergebnisspalten) — also das Ergebnisrelationsschema — bereits zur Vorübersetzungszeit bekannt ist.

Durch das Öffnen des Tupelcursors wird nun die Ausführung der Anfrage veranlasst *(Abbildung 2.4)*, wobei die aktuelle Wertebelegung der (möglicherweise in der Anfrage enthaltenen) Hostvariablen verwendet wird. Der geöffnete Tupelcursor steht anschließend *vor* dem ersten Ergebnistupel.

```
EXEC SQL OPEN unserTC;
```

Abbildung 2.4: Beispiel für das Öffnen eines Tupelcursors in statischem ESQL

Mit Hilfe der `FETCH`-Anweisung wird der Tupelcursor nun *auf* das erste Ergebnistupel positioniert, welches dadurch ins Anwendungsprogramm übertragen wird *(Abbildung 2.5)*.

In der FETCH-Anweisung müssen hierzu die Hostvariablen angegeben werden, die die einzelnen Spaltenwerte des Ergebnistupels aufnehmen sollen. In unserem Beispiel nutzen wir dafür die Variablen *knr*, *name* und *geb*. (Auf die Bedeutung der Variablen *geb_ind* werden wir im Folgenden eingehen.)

```
EXEC SQL FETCH unserTC INTO :knr, :name, :geb :geb_ind;
```

Abbildung 2.5: Beispiel für eine FETCH-Anweisung in statischem ESQL

Für jede Spalte des Anfrageergebnisses, für die Nullwerte möglich sind, wird innerhalb der FETCH-Anweisung eine zusätzliche Hostvariable (eine so genannte *Indikatorvariable*) benötigt [DD97]. Die Indikatorvariable nimmt die Information auf, ob es sich beim entsprechenden Spaltenwert des Ergebnistupels um einen Nullwert handelt. Im Falle eines Nullwerts wird der Indikatorvariable vom SQL-Laufzeitsystem eine negative Zahl zugewiesen, andernfalls 0 oder eine positive Zahl.

Da Nullwerte in unserem Beispiel ausschließlich in der Ergebnisspalte *GebDatum* vorkommen können, ist nur eine Indikatorvariable *(geb_ind)* erforderlich. Mit ihrer Hilfe lässt sich nun testen, ob anstelle des Geburtsdatums ein Nullwert zurückgeliefert wurde. In diesem Fall soll von unserem Beispielprogramm die Meldung *'Geburtsdatum unbekannt'* ausgegeben werden *(Abbildung 2.6)*.

```
cout << "Kundennummer: " << knr << "\n";
cout << "Name: " << name << "\n";
if (geb_ind < 0)
     {cout << "Geburtsdatum unbekannt \n";}
   else
     {cout << "Geburtsdatum: " << geb << "\n";}
```

Abbildung 2.6: Beispiel für die Ausgabe der Spaltenwerte des Ergebnistupels

Wir haben bisher stillschweigend vorausgesetzt, dass das Anfrageergebnis nicht leer ist und somit mindestens ein Ergebnistupel enthält. Im Allgemeinen steht jedoch erst zum Zeitpunkt der Anfrageausführung fest, ob bzw. wie viele Ergebnistupel existieren. Um flexibel auf ein Anfrageergebnis vorher unbekannter Größe reagieren zu können, bietet sich der Einsatz einer while-Schleife an *(Abbildung 2.7)*. In der Schleifenbedingung wird dabei z. B. mit Hilfe der SQLCODE-Komponente der SQLCA getestet, ob die zuletzt durchgeführte FETCH-Anweisung erfolgreich war, d. h. ein Ergebnistupel ins Anwendungsprogramm übertragen hat.

```
EXEC SQL FETCH unserTC INTO :knr, :name, :geb :geb_ind;
while (sqlca.sqlcode == 0)
{
  // hier: Verarbeitung der Werte des aktuellen Ergebnistupels
  EXEC SQL FETCH unserTC INTO :knr, :name, :geb :geb_ind;
}
```

Abbildung 2.7: Beispiel für die Anfrageergebnisverarbeitung in statischem ESQL

Sobald keine Ergebnistupel mehr ins Anwendungsprogramm zu übertragen sind, kann der Tupelcursor geschlossen werden. Hierfür stellt SQL die Anweisung `CLOSE` bereit *(Abbildung 2.8)*. Gründe für das Schließen des Tupelcursors können sein, dass das Anfrageergebnis komplett verarbeitet wurde oder dass die Abarbeitung vorzeitig beendet werden soll. Ein erneutes Öffnen des Tupelcursors (z. B. mit einer anderen Belegung der in der Anfrage enthaltenen Hostvariablen) ist möglich.

```
EXEC SQL CLOSE unserTC;
```

Abbildung 2.8: Beispiel für das Schließen eines Tupelcursors in statischem ESQL

2.4.2 Cursorkonzept und dynamisches Embedded SQL

Während das Cursorkonzept für *statisches* Embedded SQL in den verschiedenen DBMS-Produkten (z. B. DB2, Oracle und MS SQL Server) relativ einheitlich und im Wesentlichen normkonform umgesetzt wird, ist dies für *dynamisches* Embedded SQL nicht der Fall. Wir werden uns bei den folgenden Ausführungen an [Neu96] bzw. an IBMs DB2 [IBM07b, IBM07c] orientieren. Für eine Beschreibung alternativer Realisierungen bzw. des genormten Vorgehens verweisen wir auf die entsprechende Literatur, z. B. [ORA07d, DD97, ISO03c].

Auch bei dynamischem Embedded SQL wird ein Tupelcursor mit Hilfe der `DECLARE-CURSOR`-Anweisung deklariert. Dabei wird der Tupelcursor — im Gegensatz zum statischen Embedded SQL — aber nicht an eine im Quelltext „fest-verdrahtete“ `SELECT`-Anweisung gebunden, sondern lediglich an einen Anweisungsnamen. Zur Laufzeit wird dann mittels der `PREPARE`-Anweisung eine als Zeichenkette vorliegende `SELECT`-Anfrage übersetzt und dem entsprechenden Anweisungsnamen (und damit dem Tupelcursor) zugeordnet. Dieses Vorgehen wird in *Abbildung 2.9* veranschaulicht, wobei *unserStmt* als Anweisungsname und *anfr* als Hostvariable zum Aufnehmen der Zeichenkettenrepräsentation der `SELECT`-Anfrage dient.

```
EXEC SQL DECLARE unserTC CURSOR FOR unserStmt;
strcpy(anfr, "SELECT * FROM Kunde WHERE Ort='Jena'");
EXEC SQL PREPARE unserStmt FROM :anfr;
```

Abbildung 2.9: Beispiel für eine Tupelcursordeklaration in dynamischem ESQL

Die Anfrageausführung wird (wie bei statischem Embedded SQL) durch das Öffnen des Tupelcursors veranlasst *(Abbildung 2.10)*. Auch beim dynamischen Embedded SQL steht der Tupelcursor anschließend vor dem ersten Ergebnistupel (ein nicht leeres Anfrageergebnis vorausgesetzt).

```
EXEC SQL OPEN unserTC;
```

Abbildung 2.10: Beispiel für das Öffnen eines Tupelcursors in dynamischem ESQL

Beim statischen Embedded SQL stand die Struktur des Anfrageergebnisses bereits zur Vorübersetzungszeit fest. Damit konnte in der `FETCH`-Anweisung für jede Ergebnisspalte jeweils eine Hostvariable passenden Typs angegeben werden, die den entsprechenden

Spaltenwert des zu übertragenden Ergebnistupels aufnehmen sollte. Beim dynamischen Embedded SQL kann die `SELECT`-Anfrage hingegen flexibel zur Laufzeit zusammengebaut werden, sodass die Struktur des Anfrageergebnisses zur Vorübersetzungszeit i. d. R. noch unbekannt ist. Damit ist es im Allgemeinen nicht möglich, in der `FETCH`-Anweisung geeignete Hostvariablen zum Aufnehmen der Spaltenwerte anzugeben. Um trotz der zur Vorübersetzungszeit unbekannten Ergebnisstruktur mit dem Anfrageergebnis umgehen zu können, wird eine so genannte *SQLDA* (SQL Descriptor Area) eingesetzt.

Die SQLDA ist eine Datenstruktur im Anwendungsprogramm, die zur Kommunikation mit dem SQL-Laufzeitsystem genutzt wird. Wir werden die Verwendung der SQLDA im Folgenden kurz umreißen und verweisen für eine detailliertere Darstellung auf [IBM07b, IBM07c] bzw. [Neu96]. In unseren nächsten beiden Syntaxbeispielen dient die Variable *pSqlda* als Zeiger *(pointer)* auf die SQLDA.

Mit Hilfe der `DESCRIBE`-Anweisung wird zunächst die Struktur des Anfrageergebnisses ermittelt *(Abbildung 2.11)*. Das SQL-Laufzeitsystem überträgt dabei die Anzahl, die Namen sowie die Datentypen der Ergebnisspalten in entsprechende Komponenten der SQLDA. Das Anwendungsprogramm kann diese Strukturinformationen nun aus der SQLDA auslesen und (unter Berücksichtigung dieser Informationen) geeignete Speicherbereiche zur Aufnahme der Spalten- und Indikatorwerte allokieren. Die Adressen dieser allokierten Speicherbereiche werden vom Anwendungsprogramm anschließend in der SQLDA vermerkt.

```
EXEC SQL DESCRIBE unserStmt INTO :*pSqlda;
```

Abbildung 2.11: Beispiel für eine `DESCRIBE`-Anweisung in dynamischem ESQL

Analog zu statischem Embedded SQL kann eine in eine `while`-Schleife eingebettete `FETCH`-Anweisung genutzt werden, um das Anfrageergebnis tupelweise zu durchlaufen und dabei Ergebnistupel für Ergebnistupel ins Anwendungsprogramm zu übertragen *(Abbildung 2.12)*. Im Gegensatz zum statischen Fall wird in der `FETCH`-Anweisung allerdings Bezug auf die SQLDA genommen. Aus der SQLDA kann das SQL-Laufzeitsystem nun die Adressen der Speicherbereiche auslesen, in die die Spalten- bzw. Indikatorwerte übertragen werden sollen.

```
EXEC SQL FETCH unserTC USING DESCRIPTOR :*pSqlda;
while (sqlca.sqlcode == 0)
{
  // hier: Verarbeitung der Werte des aktuellen Ergebnistupels
  EXEC SQL FETCH unserTC USING DESCRIPTOR :*pSqlda;
}
```

Abbildung 2.12: Beispiel für die Anfrageergebnisverarbeitung in dynamischem ESQL

Wir möchten an dieser Stelle kurz hervorheben, dass die SQLDA für eine *beidseitige* Kommunikation zwischen Anwendungsprogramm und SQL-Laufzeitsystem genutzt wird: Beim `DESCRIBE` wird die SQLDA vom SQL-Laufzeitsystem verwendet, um dem Anwendungsprogramm die Struktur des Anfrageergebnisses mitzuteilen. Beim `FETCH` teilt das Anwendungsprogramm dem SQL-Laufzeitsystem unter Zuhilfenahme der SQLDA mit, wohin (also an welche Speicheradressen) die Spalten- und Indikatorwerte zu übertragen sind.

Wie beim statischen Embedded SQL kann der Tupelcursor geschlossen werden, sobald keine Ergebnistupel mehr ins Anwendungsprogramm zu übertragen sind *(Abbildung 2.13)*. Bei Bedarf lässt sich der (geschlossene) Tupelcursor mit Hilfe der `PREPARE`-Anweisung an eine andere `SELECT`-Anfrage binden und anschließend erneut öffnen.

```
EXEC SQL CLOSE unserTC;
```

Abbildung 2.13: Beispiel für das Schließen eines Tupelcursors in dynamischem ESQL

2.4.3 Positioniertes `UPDATE` und `DELETE`

Bei positionierten `UPDATE`s und `DELETE`s erfolgt die Auswahl der zu ändernden bzw. zu löschenden Tupel durch Bezugnahme auf die aktuelle Tupelcursorposition. Damit ein positioniertes Ändern oder Löschen prinzipiell zulässig ist, muss die an den Tupelcursor gebundene `SELECT`-Anfrage verschiedene Voraussetzungen erfüllen. Insbesondere muss jedem Tupel des Anfrageergebnisses jeweils genau *ein* Tupel der zugrunde liegenden Basistabelle zugeordnet sein, was impliziert, dass die `SELECT`-Anweisung beispielsweise keine `DISTINCT`-Klausel enthalten darf. Für eine umfassende Auflistung aller geforderten Bedingungen verweisen wir auf [DD97].

Ist bekannt, dass keine positionierten `UPDATE`s und `DELETE`s durchgeführt werden sollen, können diese mit Hilfe der `FOR-READ-ONLY`-Klausel bei der Tupelcursordeklaration explizit ausgeschlossen werden. Sind positionierte Änderungs- und Löschanweisungen ohnehin nicht möglich (weil die `SELECT`-Anfrage z. B. das Schlüsselwort `DISTINCT` enthält), ist die Angabe der `FOR-READ-ONLY`-Klausel ohne Bewandtnis, andernfalls kann durch diese Klausel u. U. eine Performancesteigerung erreicht werden [IBM07b].

Positionierte `UPDATE`s und `DELETE`s sind erlaubt, wenn sie (auf Grund der Eigenschaften der `SELECT`-Anfrage) prinzipiell zulässig sind und sie nicht explizit mittels `FOR READ ONLY` ausgeschlossen wurden. Die explizite Angabe einer `FOR-UPDATE`-Klausel bei der Tupelcursordeklaration ist *keine* notwendige Voraussetzung für die Ausführbarkeit eines positionierten `UPDATE`s.

Bei der Durchführung eines positionierten `UPDATE`s oder `DELETE`s muss der (geöffnete) Tupelcursor auf einem Ergebnistupel positioniert sein. Geändert bzw. gelöscht wird dann das mit diesem Ergebnistupel korrespondierende Tupel der zugrunde liegenden Basistabelle. Obwohl bei der Tupelauswahl Bezug auf die Tupelcursorposition im *Anfrageergebnis* genommen wird, werden die Änderungs- bzw. Löschoperationen also auf der *Basistabelle* ausgeführt, gegen die die (an den Tupelcursor gebundene) Anfrage gestellt wurde [DD97]. Bei jedem (erfolgreichen) positionierten `UPDATE` oder `DELETE` wird jeweils genau *ein* Tupel der Basistabelle geändert bzw. gelöscht.

Die *Abbildungen 2.14* und *2.15* zeigen Beispiele für eine positionierte Änderungs- bzw. Löschanweisung. Wir setzen dabei voraus, dass der Tupelcursor *unserTC* (wie in den vorigen beiden Abschnitten beschrieben) an eine Anfrage gegen die Tabelle *Kunde* gebunden ist. Ferner sei der Tupelcursor zum Zeitpunkt der Anweisungsausführung auf ein bestimmtes Tupel des Anfrageergebnisses positioniert. Das diesem Ergebnistupel zugrunde liegende Tupel der Kundentabelle wird entsprechend geändert *(hier: Erfurt wird neuer Wohnort des Kunden)* bzw. gelöscht.

```
UPDATE Kunde SET Ort='Erfurt' WHERE CURRENT OF unserTC
```

Abbildung 2.14: Beispiel für ein positioniertes `UPDATE`

```
DELETE FROM Kunde WHERE CURRENT OF unserTC
```

Abbildung 2.15: Beispiel für ein positioniertes `DELETE`

Positionierte `UPDATE`s und `DELETE`s sind sowohl in statischem als auch in dynamischem Embedded SQL erlaubt. Die in den Abbildungen 2.14 und 2.15 aufgeführten Anweisungen könnten also (mit dem Präfix `EXEC SQL` markiert) „fest-verdrahtet“ im Quelltext stehen oder dynamisch zur Laufzeit (als Zeichenketten) zusammengebaut werden.

2.5 LOB-Locator-Prinzip

Die Bezeichnung *LOB (large object)* dient als Sammelbegriff für so genannte LOB-Datentypen wie beispielsweise *BLOB (binary large object)* oder *CLOB (character large object)* [DD97]. LOB-Werte (deren innere Struktur dem Datenbanksystem nicht bekannt ist) können sehr groß sein — typischerweise bis zu 2 oder 4 Gigabyte [MB06].

Beim Weiterbewegen des Tupelcursors (SQL-`FETCH`-Anweisung) werden die einzelnen Spaltenwerte des nun aktuellen Ergebnistupels normalerweise *komplett* ins Anwendungsprogramm übertragen (Abschnitt 2.4). Aufgrund ihrer Größe ist dieses Vorgehen für LOB-Werte oftmals jedoch nicht erwünscht. Einen Ausweg bietet das LOB-Locator-Prinzip.

Ein *LOB-Locator* ist eine Hostvariable, die beim `FETCH` nicht den (u. U. sehr großen) LOB-Wert aufnimmt, sondern nur einen Verweis auf diesen. Der LOB-Locator kann anschließend in anderen Embedded-SQL-Anweisungen genutzt werden, um z. B. (kleinere) Ausschnitte des entsprechenden LOB-Werts auszuwählen und ins Anwendungsprogramm zu übertragen.

Das (von SQL-Norm und Produkten unterstützte) LOB-Locator-Prinzip ist also ein Mechanismus, der es dem Anwendungsprogramm erlaubt, mit einem im Anfrageergebnis enthaltenen LOB-Wert zu arbeiten, ohne dass dieser komplett ins Anwendungsprogramm übertragen werden muss. Dabei wird im Anwendungsprogramm ein vom Datenbanksystem verwalteter Verweis auf den datenbanksytemseitig bereitgestellten LOB-Wert genutzt.

Bei Anfrageergebnissen, die LOB-Werte enthalten, wird i. d. R. mit LOB-Locatoren gearbeitet. Alternativ ist es jedoch auch möglich, auf den Einsatz von LOB-Locatoren zu verzichten und die LOB-Werte *komplett* ins Anwendungsprogramm zu übertragen. Die Entscheidung zwischen beiden Varianten wird im statischen Embedded SQL bei der Deklaration der Hostvariablen getroffen. Beispielsweise fungieren Hostvariablen vom Typ `SQL TYPE IS CLOB AS LOCATOR` als LOB-Locatoren, während Hostvariablen vom Typ `SQL TYPE IS CLOB` den kompletten LOB-Wert aufnehmen. Im dynamischen Embedded SQL teilt das Anwendungsprogramm dem SQL-Laufzeitsystem mit Hilfe der SQLDA mit, welche Variante zu nutzen ist. Damit kann man der `FETCH`-Anweisung weder bei statischem noch bei dynamischem Embedded SQL ansehen, ob mit LOB-Locatoren gearbeitet wird oder ob die kompletten LOB-Werte übertragen werden.

2.6 XML

Die Metaauszeichnungssprache XML *(eXtensible Markup Language)* wurde 1998 vom World Wide Web Consortium *(W3C)* als offizielle Empfehlung *(W3C Recommendation)* verabschiedet. XML [Von05, HM05] entstand aus der 1986 von der ISO genormten Metaauszeichnungssprache SGML *(Standard Generalized Markup Language)* [ISO86], deren Wurzeln bis in die 1970er Jahre zurückreichen.

Die im Februar 1998 veröffentlichte (ursprüngliche) Version *XML 1.0* [W3C98b, NT05] gilt auch heutzutage noch. Aufgrund einiger notwendig gewordener (kleinerer) Korrekturen verabschiedete das W3C in den Jahren 2000 [W3C00], 2004 [W3C04b] und 2006 [W3C06a] zwar aktualisierte Fassungen der XML-Spezifikation — es handelt sich dabei aber *nicht* um neuere XML-Versionen.

Seit Februar 2004 ist auch *XML 1.1* eine offizielle W3C-Empfehlung. XML 1.1 [W3C04c, W3C06b] soll XML 1.0 allerdings nicht ablösen, sondern wurde als Alternative konzipiert, deren wesentliche Neuerung darin besteht, eine höhere Flexibilität bezüglich der verwendbaren Zeichen zu gewährleisten. Während XML 1.0 auf der aus dem Jahr 1996 stammenden Unicode-Version 2.0 [Con96] basiert, berücksichtigt XML 1.1 die permanente Weiterentwicklung des Unicode-Standards [Con00, Con03, Con06]. Da die Unterschiede zwischen XML 1.0 und XML 1.1 für den weiteren Verlauf der vorliegenden Arbeit jedoch nicht relevant sind, werden wir bei allen folgenden Betrachtungen auf eine Unterscheidung zwischen XML 1.0 und XML 1.1 verzichten.

Unter dem Begriff „XML“ wird im Allgemeinen nicht nur die eigentliche XML-Spezifikation verstanden — „XML“ dient vielmehr als Sammelbegriff für eine Vielzahl von „XML-nahen“ Technologien, wie beispielsweise XInclude [W3C06d], XPointer [W3C03], XLink [W3C01c], XBase [W3C01a], SOAP [W3C07a], WSDL [W3C07b], XSLT [W3C99c], XML-Namensräume [W3C06c], XML-Schema [W3C04e], XML-Infoset [W3C04d], DOM [W3C04a], SAX [MB02], XPath [W3C07c] oder XQuery [W3C07d]. Da eine Einführung in sämtliche XML-bezogene Themen den Rahmen dieser Arbeit sprengen würde, betrachten wir im Folgenden nur jene Konzepte, die für das Verständnis der weiteren Arbeit wichtig sind. Hierzu zählen XML-Dokumente (Abschnitt 2.7.1), XML-Namensräume (Abschnitt 2.7.2), XML-Schema (Abschnitt 2.7.3), das XML-Infoset (Abschnitt 2.7.4), DOM (Abschnitt 2.8), XPath bzw. XQuery (Abschnitt 2.9) sowie das XQuery-Datenmodell (Abschnitt 2.10).

XML-basierte Techniken haben im Verlauf der letzten Jahre enorm an Bedeutung gewonnen. In vielen Anwendungsszenarien werden sie zur Repräsentation, zum Austausch, zur Speicherung bzw. zur Verarbeitung von (semi-)strukturierten Daten eingesetzt. Damit ist XML aus der heutigen IT-Landschaft nicht mehr wegzudenken.

2.7 XML-Basiskonzepte

Wir werden in diesem Abschnitt kurz diejenigen Konzepte vorstellen, die im Allgemeinen zum Kern der XML-Technologiefamilie gezählt werden [Von05]. Für eine ausführlichere Einführung in die entsprechenden Themen verweisen wir auf die einschlägige Fachliteratur, z. B. [Von05, HM05, EE03].

2.7.1 XML-Dokumente

Im Mittelpunkt der (eigentlichen) XML-Spezifikation [W3C06a] steht die Definition so genannter *XML-Dokumente*. Das grundlegende Konstrukt eines XML-Dokuments ist das *Element*. Ein Element besteht aus einem *Start-Tag*, dem Elementinhalt und einem *End-Tag*. Sowohl Start- als End-Tag enthalten den Namen des Element(typ)s. Als Elementinhalt sind Text, (Kind-)Elemente oder eine Mischung aus beidem zulässig. Elemente ohne Inhalt sind ebenfalls erlaubt. Für sie gibt es eine abkürzende Schreibweise, bei der anstelle von Start- und End-Tag ein *Leeres-Element-Tag* verwendet wird. Die *Tags*[1] (welche mit einer öffnenden spitzen Klammer beginnen und mit einer schließenden spitzen Klammer enden) dienen also zur Kennzeichnung der Elemente.

Das XML-Dokument in unserem Beispiel *(Abbildung 2.16)* besteht aus einem Element namens *Einkauf*, welches als Elementinhalt zwei Kindelemente mit dem Namen *Artikel* besitzt. Jedes *Artikel*-Element enthält wiederum jeweils die Elemente *Bezeichnung* und *Preis*, bei deren Inhalt es sich um Text handelt.

```
<Einkauf Datum="03.02.2008" Filiale="Jena">
   <Artikel Anzahl="2">
      <Bezeichnung>Kerze</Bezeichnung>
      <Preis MWSt-Satz="19.0">5.07</Preis>
   </Artikel>
   <Artikel Anzahl="1">
      <Bezeichnung>Torte</Bezeichnung>
      <Preis MWSt-Satz="7.0">21.05</Preis>
   </Artikel>
   <!--Kassierer 259-->
</Einkauf>
```

Abbildung 2.16: Beispiel für ein XML-Dokument

Ein XML-Dokument besitzt *genau ein* Element, welches nicht in anderen Elementen enthalten ist. Dieses Element wird als *Wurzelelement* bezeichnet und beinhaltet sämtliche anderen Elemente des XML-Dokuments. In unserem Beispiel ist *Einkauf* das Wurzelelement.

Elementen können *Attribute* zugeordnet sein, die jeweils aus einem Namen und einem Wert bestehen. In unserem XML-Dokument besitzt das Element *Einkauf* beispielsweise die beiden Attribute *Datum* und *Filiale* mit den Werten *03.02.2008* bzw. *Jena*. XML-Dokumente dürfen außerdem *Kommentare* enthalten. In Abbildung 2.16 handelt es sich z. B. bei *<!–Kassierer 259–>* um einen Kommentar.

Als weitere Bestandteile eines XML-Dokuments sind auch eine einleitende *XML-Deklaration*, eine *Document Type Definition (DTD)* sowie *Verarbeitungsanweisungen* zulässig. Für eine detaillierte Erläuterung dieser Komponenten verweisen wir auf die entsprechende Fachliteratur, z. B. [HM05, Von05].

XML-Dokumente, die den in der XML-Spezifikation festgelegten Syntaxregeln genügen, werden als *wohlgeformt (well-formed)* bezeichnet. Diese Regeln erzwingen u. a., dass genau ein Wurzelelement existiert, dass es zu jedem Start-Tag ein zugehöriges End-Tag gibt

[1] *engl. für* Etiketten, Schildchen

und dass die Elemente korrekt ineinander geschachtelt sind. Die Bezeichnungen *XML-Dokument* und *wohlgeformtes XML-Dokument* sind allerdings synonym, da es per Definition keine nicht-wohlgeformten XML-Dokumente gibt. Wir werden später jedoch sehen, dass der Begriff *XML-Dokument* in anderen Spezifikationen teils weiter gefasst ist als hier. Um Verwechslungen zu vermeiden, werden wir die „wahren“ XML-Dokumente (also die, die der XML-Spezifikation entsprechen) als *wohlgeformte XML-Dokumente* bezeichnen.

Mittels eines *XML-Schemas* kann die für ein XML-Dokument als zulässig erachtete Struktur eingeschränkt werden. So lässt sich beispielsweise festlegen, welche Elemente das XML-Dokument enthalten darf bzw. muss. Genügt ein (wohlgeformtes) XML-Dokument dem vorgegebenen XML-Schema, nennt man es *gültig (valid)*. Wir werden hierauf im Abschnitt 2.7.3 näher eingehen.

2.7.2 XML-Namensräume

In einem XML-Dokument werden i. d. R. Element- und Attributnamen verwendet, die zum entsprechenden Anwendungsgebiet passen. In unserem XML-Dokument aus Abbildung 2.16 gibt es beispielsweise *Artikel*-Elemente, welche Einkaufsartikel wie Torten oder Kerzen repräsentieren. Das Attribut *Anzahl* gibt dabei an, wie oft der jeweilige Artikel gekauft wurde. Dieselben Element- und Attributnamen können in anderen Kontexten jedoch mit einer völlig abweichenden Bedeutung benutzt werden. So kann ein *Artikel*-Element auch für einen Zeitungsartikel stehen, während das *Anzahl*-Attribut die Anzahl der am Artikel beteiligten Autoren angibt.

Werden in ein und demselben XML-Dokument Element- und Attributnamen aus verschiedenen Anwendungskontexten genutzt, kann es zu Namenskollisionen kommen. Bezogen auf unser Beispiel wäre beim Auftreten eines *Artikel*-Elements nicht entscheidbar, ob es sich um einen Einkaufsartikel oder um einen Zeitungsartikel handelt. Um derartige Namenskonflikte zu verhindern, wurden vom W3C im Jahr 1999 die so genannten *XML-Namensräume* [W3C99a, W3C06c, HM05] eingeführt. Gleichnamige Elemente bzw. Attribute mit unterschiedlichen Bedeutungen können nun verschiedenen XML-Namensräumen zugeordnet werden und sind damit voneinander unterscheidbar.

Namensräume werden durch eine URI *(Uniform Resource Identifier)* [BLFM05] identifiziert, wobei i. d. R. eine URL *(Uniform Resource Locator)* genutzt wird. Beispielsweise könnten unsere *Artikel*-Elemente aus Abbildung 2.16 an den Namensraum *http://www.meineURL.de/Einkauf* gebunden werden, während für die *Artikel*-Elemente aus dem Presse-Kontext der Namensraum *http://www.meineURL.de/Presse* verwendet wird. Die entsprechenden URLs müssen hierbei jedoch *nicht* tatsächlich existieren — URLs werden lediglich eingesetzt, um *eindeutige* Namensräume zu gewährleisten.

Die Kombination aus URL (z. B. *http://www.meineURL.de/Einkauf*) und ursprünglichem Namen (hier: *Artikel*) wird *erweiterter Name (expanded name)* genannt. Der URL-Bestandteil des erweiterten Namens heißt *Namensraumname*, der ursprüngliche Name wird als *lokaler Name* bezeichnet. Zwei erweiterte Namen gelten nur dann als gleich, wenn sie in beiden Namensbestandteilen jeweils übereinstimmen. Unsere *Artikel*-Elemente aus Abbildung 2.16 und die *Artikel*-Elemente aus dem Presse-Kontext besitzen damit *verschiedene* erweiterte Namen, da sie sich im Namensraumnamen unterscheiden.

URLs sind oftmals sehr lang und können außerdem Zeichen enthalten, die in Element- und Attributnamen nicht erlaubt sind. Aus diesen Gründen werden in XML-Dokumenten statt

erweiterter Namen so genannte *qualifizierte Namen (qualified names)* verwendet. Ein qualifizierter Name (z. B. *thm:Artikel*) besteht aus einem optionalen *Namensraumpräfix* (hier: *thm*) und dem lokalen Namen (hier: *Artikel*). Der Namensraumpräfix, der durch einen Doppelpunkt vom lokalen Namen getrennt wird, dient dabei als Kürzel für einen Namensraumnamen (z. B. *http://www.meineURL.de/Einkauf*). Die Bindung des Namensraumkürzels an den Namensraumnamen erfolgt Attribut-basiert, wie in *Abbildung 2.17* angedeutet. Außerdem ist die Deklaration eines Default-Namensraums möglich, bei dem auf die Nutzung eines Namensraumpräfixes verzichtet wird. Für detailliertere Erklärungen verweisen wir auf die entsprechende Literatur, z. B. [HM05, W3C06c].

```
<thm:Artikel xmlns:thm="http://www.meineURL.de/Einkauf" ...>
...
</thm:Artikel>
```

Abbildung 2.17: Beispiel für eine Namensraumdeklaration

2.7.3 XML-Schema

In wohlgeformten XML-Dokumenten dürfen prinzipiell beliebige Elemente und Attribute mit (fast) beliebigen Elementinhalten bzw. Attributwerten auftreten. Um XML-Dokumente jedoch geeignet verarbeiten zu können, ist es i. d. R. erforderlich, Einschränkungen bezüglich der verwendbaren Elemente bzw. Attribute vorzunehmen. Hierfür hat das W3C im Jahr 2001 die XML-Schemabeschreibungssprache *XML-Schema* [W3C01d, W3C04e, W3C04f, W3C04g] verabschiedet.

Mittels XML-Schema [LS04b, EE03] kann u. a. festgelegt werden, welche Elemente das XML-Dokument enthalten darf, in welcher Reihenfolge bzw. Anzahl die Elemente auftreten dürfen, welche Elementinhalte zulässig sind, welche Attribute zu welchen Elementen gehören und welche Datentypen die Attributwerte besitzen sollen. XML-Schema geht dabei in seinen Möglichkeiten weit über die *DTD*s hinaus, die im Rahmen der XML-1.0-Spezifikation zur Schemabeschreibung vorgesehen waren [HM05, EE03]. Außerdem überwindet XML-Schema etliche Defizite, mit denen DTDs behaftet waren [LS04b]. XML-Schema kann damit als Nachfolger der DTDs angesehen werden [Von05].

Um Missverständnisse zu vermeiden, möchten wir darauf hinweisen, dass der Begriff *XML-Schema* doppelt belegt ist. Zum einen steht XML-Schema (wie bereits erwähnt) für die vom W3C verabschiedete XML-Schemabeschreibungssprache. Zum anderen wird ein konkretes Schema (welches mit Hilfe dieser Schemabeschreibungssprache formuliert wurde) ebenfalls als XML-Schema bezeichnet. Wenn wir den XML-Schema-Begriff im Laufe der weiteren Arbeit verwenden, ergibt sich die jeweils zutreffende Bedeutung i. d. R. allerdings aus dem Zusammenhang, sodass wir auf die Einführung einer (künstlichen) Begriffstrennung verzichten können.

Bei einem (konkreten) XML-Schema handelt es sich übrigens selbst um ein wohlgeformtes XML-Dokument. Die Nutzung der XML-Syntax erleichtert die Verwaltung und Verarbeitung der XML-Schemata, da dabei auf (vorhandene) XML-Techologien zurückgegriffen werden kann.

Das Typkonzept von XML-Schema unterscheidet zwischen *einfachen (Daten-)Typen (simple types)* und *komplexen (Daten-)Typen (complex types)* [LS04b, W3C04e]. Einfache Ty-

pen lassen weder Attribute noch Kindelemente zu und werden zur Beschreibung von Elementen und Attributen genutzt. Komplexe Typen können hingegen das Auftreten von Attributen und Kindelementen erlauben und werden zur Charakterisierung von Elementen eingesetzt. Während Elemente also einen einfachen oder einen komplexen Typ besitzen dürfen, sind bei Attributen nur einfache Typen möglich.

XML-Schema stellt eine Reihe vordefinierter einfacher Typen bereit. Hierzu zählen numerische Datentypen wie *decimal, integer* oder *byte*, der Zeichenkettendatentyp *string*, Datentypen für Datums- bzw. Zeitangaben wie *date, time* oder *dateTime* sowie etliche weitere Typen. Für eine detailliertere Beschreibung der einzelnen Datentypen verweisen wir auf [W3C04g].

Ausgehend von den vordefinierten einfachen Datentypen lassen sich durch Einschränkung, Vereinigung oder Listenbildung eigene einfache Typen bilden [LS04b]. Durch Einschränkung des Typs *string* ist es beispielsweise möglich, einen Zeichenkettendatentyp zu konstruieren, der für zulässige Zeichenketten eine gewisse Minimal- bzw. Maximallänge fordert. Als Illustration hierfür zeigt *Abbildung 2.18* die Definition des (einfachen) Datentyps *unserString*, der Zeichenketten mit einer Länge von 5 bis 21 Zeichen erlaubt.

```
<xs:simpleType name="unserString">
   <xs:restriction base="xs:string">
      <xs:minLength value="5"/>
      <xs:maxLength value="21"/>
   </xs:restriction>
</xs:simpleType>
```

Abbildung 2.18: Beispiel für eine Typdefinition durch Einschränkung

Bei komplexen Typen (die im Gegensatz zu einfachen Typen Attribute und Kindelemente zulassen können) unterscheidet man in Abhängigkeit davon, ob sie Kindelemente und/oder textuelle Inhalte erlauben, zwischen vier Unterarten [W3C04f]: Komplexe Typen mit *einfachem Inhalt* lassen zwar textuellen Inhalt zu, nicht jedoch Kindelemente. Bei komplexen Typen mit *leerem Inhalt* sind weder textuelle Inhalte noch Kindelemente erlaubt. Komplexe Typen mit *gemischtem Inhalt* ermöglichen ein gemeinsames Auftreten von textuellen Inhalten und Kindelementen. Bei komplexen Typen mit *Nur-Element-Inhalt* sind ausschließlich Kindelemente, nicht jedoch textuelle Inhalte zulässig.

Die zuvor beschriebene Klassifizierung der komplexen Typen ist sowohl bei der Bestimmung des getypten Werts von XQuery-Element-Knoten (Abschnitt 2.10.2.5) als auch bei der Ermittlung der typed-value-orientierten Repräsentation von XQuery-Sequenzen (Kapitel 3) relevant. Wir werden die vier Unterarten der komplexen Typen deshalb später wieder aufgreifen. Zwei dieser Unterarten wollen wir im Folgenden noch kurz anhand unseres Beispiel-XML-Dokuments aus Abschnitt 2.7.1 betrachten.

Unsere *Preis*-Elemente aus Abbildung 2.16 besitzen ein Attribut. Möchten wir diesen Elementen nun einen Typ zuordnen, muss es sich folglich um einen *komplexen* Typ handeln. Da im Elementinhalt zwar Text, aber kein Kindelement vorkommt, kann ein komplexer Typ mit *einfachem Inhalt* genutzt werden. *Abbildung 2.19* zeigt die Definition eines entsprechenden Typs namens *PreisTyp*. Dieser Typ entsteht durch die Erweiterung des vordefinierten einfachen Typs *decimal* um das Attribut *MWSt-Satz*.

```
<xs:complexType name="PreisTyp">
   <xs:simpleContent>
      <xs:extension base="xs:decimal">
         <xs:attribute name="MWSt-Satz" type="xs:decimal"/>
      </xs:extension>
   </xs:simpleContent>
</xs:complexType>
```

Abbildung 2.19: Beispiel für die Definition eines kompl. Typs mit einfachem Inhalt

Soll nun auch unseren *Artikel*-Elementen ein Typ zugeordnet werden, so ist dabei (wegen des Attributs und der Kindelemente) ebenfalls ein *komplexer* Typ zu verwenden. Da als Elementinhalt diesmal aber ausschließlich Kindelemente (und keine textuellen Inhalte) zulässig sein sollen, bietet sich der Einsatz eines komplexen Typs mit *Nur-Element-Inhalt* an. *Abbildung 2.20* zeigt eine entsprechende Typdefinition. Dem Attribut *Anzahl* wird dabei der vordefinierte einfache Typ *integer* zugewiesen. Als Elementinhalt wird eine Folge, bestehend aus den Elementen *Bezeichnung* und *Preis*, festgelegt. Das *Bezeichnung*s-Element besitzt hierbei den von uns definierten einfachen Typ *unserString*, während dem *Preis*-Element der von uns definierte komplexe Typ *PreisTyp* zugeordnet wird.

```
<xs:complexType name="ArtikelTyp">
   <xs:sequence>
      <xs:element name="Bezeichnung" type="unserString"/>
      <xs:element name="Preis" type="PreisTyp"/>
   </xs:sequence>
   <xs:attribute name="Anzahl" type="xs:integer"/>
</xs:complexType>
```

Abbildung 2.20: Beispiel für die Definition eines kompl. Typs mit Nur-Element-Inhalt

Analog ließe sich nun auch für das *Einkauf*s-Element aus unserem Beispiel ein geeigneter komplexer Typ mit Nur-Element-Inhalt definieren. Auf eine explizite Darstellung dieser Typdefinition verzichten wir jedoch aus Platzgründen. Dass das *Einkauf*s-Element außer den *Artikel*-Elementen auch einen Kommentar enthält, braucht bei der Schemafestlegung übrigens nicht berücksichtigt zu werden: Kommentare (und Verarbeitungsanweisungen) werden von XML-Schema nicht erfasst.

Bisher haben wir betrachtet, wie mittels XML-Schema Datentypen für Elemente und Attribute festgelegt werden können. Darüber hinaus bietet XML-Schema aber auch die Möglichkeit, Schlüssel- bzw. Fremdschlüsselbedingungen zu formulieren: Mit Hilfe von *unique* lässt sich die Eindeutigkeit von Attribut- und Elementwerten innerhalb eines bestimmten Dokumentbereichs fordern. Soll außer der Eindeutigkeit auch die Existenz der entsprechenden Werte erzwungen werden, so ist dies mittels *key* möglich. Zur Festlegung von Fremdschlüsselbeziehungen steht *keyref* zur Verfügung. Für weitere Details verweisen wir auf die einschlägige Fachliteratur, z. B. [LS04b, EE03].

Genügt ein XML-Dokument den Anforderungen eines konkreten XML-Schemas, so wird es als *gültig* bezüglich dieses XML-Schemas bezeichnet. Der Vorgang der Gültigkeitsprüfung heißt *Validierung*. Wir werden die Validierungsthematik bei der Betrachtung des XQuery-Datenmodells (Abschnitt 2.10) erneut aufgreifen.

2.7.4 XML-Infoset

Die XML-1.0-Spezifikation erlaubt teilweise syntaktische Unterscheidungen, die aus semantischer Sicht nicht relevant sind. Beispielsweise kann ein leeres Element sowohl mittels Start- und End-Tag *(<Kreditkarten></Kreditkarten>)* als auch mit Hilfe eines einzelnen Tags *(<Kreditkarten/>)* formuliert werden. Obwohl sich beide Varianten syntaktisch unterscheiden, stehen sie doch beide für das Gleiche, nämlich für ein leeres Element namens *Kreditkarten*.

XML-Dokumente enthalten also zum einen relevante Informationen (wie z. B. die Existenz eines leeren *Kreditkarten*-Elements) und zum anderen unwichtige Details (wie die Nutzung der konkreten Syntaxvariante). Dies wirft die Frage auf, welche Informationen eines XML-Dokuments als *relevant* erachtet werden und wie sich diese Informationen formal beschreiben lassen. Eine Antwort auf diese Frage liefert das *XML Information Set* [W3C01b, W3C04d], welches im Jahr 2001 vom W3C verabschiedet wurde.

Ein (in der zuvor genannten W3C-Empfehlung definiertes) *Infoset* ist eine abstrakte Repräsentation der *wesentlichen* Inhalte eines XML-Dokuments [MB06]. Zu diesen wesentlichen Informationen zählt beispielsweise, welche Elemente, Attribute, Kommentare oder Verarbeitungsanweisungen das XML-Dokument enthält, welches Attribut zu welchem Element gehört, welche Kindelemente ein Element besitzt usw. Als unwichtige (und vom Infoset damit nicht repräsentierte) Details gelten u. a. die Attributreihenfolge innerhalb eines Elements oder die konkrete Syntaxvariante, die zur Formulierung eines leeren Elements genutzt wurde.

Ein Infoset setzt sich aus verschiedenen *Informationseinheiten (information items)* zusammen, welche jeweils einen bestimmten Teil des XML-Dokuments beschreiben. So wird beispielsweise jedes Element durch eine *Element-Informationseinheit* repräsentiert. Ein Infoset für unser Beispiel-XML-Dokument aus Abschnitt 2.7.1 würde somit also sieben Element-Informationseinheiten enthalten. Jede Element-Informationseinheit besitzt verschiedene *Eigenschaften (properties)*, die das entsprechende Element näher beschreiben. Zu diesen Eigenschaften zählen u. a. der lokale Elementname, der Namensraumname, die Menge der zum Element gehörenden Attribute sowie die Liste der Kinder.

Auch für Attribute, Kommentare und Verarbeitungsanweisungen gibt es entsprechende Informationseinheiten mit passenden Eigenschaften. Insgesamt werden vom XML Information Set elf Arten von Informationseinheiten unterschieden. Für weitere Details verweisen wir auf die einschlägige Literatur, z. B. [MB06], bzw. die W3C-Empfehlung [W3C04d].

2.8 DOM

Das *Document Object Model (DOM)* ist eine Programmierschnittstelle (API), die für den lesenden und ändernden Zugriff auf XML-Dokumente genutzt werden kann [Rei05]. Die erste Fassung von DOM *(DOM Level 1)* wurde 1998 vom W3C als offizielle Empfehlung verabschiedet [W3C98a]. Die derzeit aktuelle DOM-Spezifikation *(DOM Level 3)* stammt aus dem Jahr 2004 [W3C04a].

Ein zu verarbeitendes XML-Dokument wird als Baum *(DOM-Baum)* aufgefasst, dessen Knoten mit bestimmten Teilen des XML-Dokuments korrespondieren. Insgesamt werden zwölf Knotenarten unterschieden, die in etwa den Arten der Informationseinheiten des

XML-Infosets (Abschnitt 2.7.4) entsprechen [MB06]. Alle Elemente, Attribute, Kommentare, Verarbeitungsanweisungen und textuellen Inhalte des XML-Dokuments werden im DOM-Baum durch *Element-*, *Attribut-*, *Kommentar-*, *Verarbeitungsanweisungs-* bzw. *Textknoten* repräsentiert. Außer den textuellen Inhalten von Elementen werden dabei auch die Attributwerte als eigenständige Textknoten dargestellt. Ein DOM-Baum für unser Beispiel-XML-Dokument aus Abschnitt 2.7.1 enthielte somit also sieben Elementknoten, sechs Attributknoten, einen Kommentarknoten und zehn Textknoten.

Anwendungsprogramme können mit Hilfe diverser *DOM-Methoden* durch den DOM-Baum navigieren. Ausgehend von einem Elementknoten ist es mittels der *getAttributeNode()*-Methode beispielsweise möglich, zu einem bestimmten Attributknoten zu gelangen. Auch ein Aufsuchen des Vaterknotens, des ersten bzw. letzten Kindknotens oder des linken bzw. rechten Geschwisterknotens wird unterstützt.

Bei der Navigation durch den DOM-Baum können Informationen ausgelesen werden, die dann für eine Weiterverarbeitung im Anwendungsprogramm zur Verfügung stehen. Außer dem Auslesen von Informationen besteht aber auch die Möglichkeit, Änderungen am DOM-Baum vorzunehmen — so lassen sich beispielsweise Knoten einfügen, ersetzen und löschen [KM02]. DOM stellt hierfür die Methoden *appendChild()*, *replaceChild()* und *removeChild()* bereit. Für eine umfassende Übersicht über alle DOM-Methoden verweisen wir auf die Spezifikation [W3C04a].

2.9 XPath und XQuery

Die erste Version der *XML Path Language* — kurz *XPath 1.0* — wurde im Jahr 1999 als offizielle W3C-Empfehlung verabschiedet [W3C99b]. Das primäre Ziel von XPath 1.0 besteht darin, Teile eines XML-Dokuments adressieren bzw. auswählen zu können. XPath 1.0 kann damit als Anfragesprache für XML-Dokumente angesehen werden [MB06].

In XPath 1.0 wird ein XML-Dokument als Baum aufgefasst, welcher aus Knoten unterschiedlicher Knotenarten besteht. Im Gegensatz zu DOM (Abschnitt 2.8) wird dabei aber nur zwischen sieben (statt zwölf) Knotenarten unterschieden. Wie auch DOM unterstützt XPath 1.0 u. a. Elementknoten, Attributknoten, Kommentarknoten, Verarbeitungsanweisungsknoten und Textknoten.

Zur Auswahl von XML-Dokument-Teilen (d. h. zur Anfrage gegen XML-Dokumente) werden *XPath-1.0-Ausdrücke* genutzt. Die wichtigste Form eines XPath-1.0-Ausdrucks ist der *Pfadausdruck (path expression)*, auch bekannt als *Lokalisierungspfad (location path)*. Ein Lokalisierungspfad besteht aus einem oder mehreren *Lokalisierungsschritten (location steps)*, die durch Schrägstriche *(/)* voneinander getrennt sind. Wir werden im Folgenden kurz ein kleines Beispiel für einen Lokalisierungspfad betrachten und verweisen für umfassendere Erläuterungen auf [MB06] bzw. [W3C99b].

Sollen mittels XPath 1.0 beispielsweise die Bezeichnungen der doppelt gekauften Artikel ermittelt werden, so ist dies mit dem in *Abbildung 2.21* gezeigten Pfadausdruck möglich. Ausgehend vom Wurzelelement *Einkauf* unseres XML-Dokuments aus Abschnitt 2.7.1 (erster Lokalisierungsschritt) werden im zweiten Lokalisierungsschritt alle *Artikel*-Kindelemente ermittelt, deren *Anzahl*-Attribut den Wert *2* besitzt. Im dritten und zugleich letzten Lokalisierungsschritt werden von diesen *Artikel*-Elementen die *Bezeichnung*s-Kindelemente ausgewählt und als Anfrageergebnis zurückgeliefert.

```
/Einkauf/Artikel[@Anzahl="2"]/Bezeichnung
```

Abbildung 2.21: Beispiel für einen Pfadausdruck

Im Jahr 2007 wurde *XPath 2.0* [W3C07c] vom W3C als offizielle Empfehlung verabschiedet. Bei XPath 2.0 handelt es sich um eine Weiterentwicklung von XPath 1.0, die eine deutlich erweiterte Funktionalität aufweist [MB06]. Die wohl wichtigste Neuerung gegenüber XPath 1.0 liegt im Wechsel des zugrunde gelegten Datenmodells. Während XPath 1.0 auf der baumbasierten Darstellung von XML-Dokumenten beruht, operiert XPath 2.0 auf *Sequenzen* des *XQuery-Datenmodells* [W3C07e]. Wir werden dieses Datenmodell ausführlich im Abschnitt 2.10 betrachten. Für eine umfassende Beschreibung von XPath 2.0 und eine detaillierte Gegenüberstellung mit XPath 1.0 verweisen wir auf die Literatur, z. B. [MB06].

Zeitgleich mit XPath 2.0 verabschiedete das W3C auch *XQuery 1.0* [W3C07d] als offizielle Empfehlung. XQuery 1.0 [LS04b, LS04a] wurde als Erweiterung von XPath 2.0 konzipiert und gilt derzeit als die wichtigste und leistungsfähigste Anfragesprache für XML-basierte Datenbestände. Wie auch XPath 2.0 operiert XQuery auf Sequenzen des XQuery-Datenmodells. Neben den aus XPath bekannten Pfadausdrücken gelten die so genannten *FLWOR-Ausdrücke* [LS04b] als wichtigstes Sprachkonstrukt von XQuery. Mit Hilfe dieser FLWOR-Ausdrücke lassen sich sehr komplexe XQuery-Anfragen formulieren. Für nähere Details und eine umfassende Einführung in XQuery verweisen wir auf die einschlägige Fachliteratur, z. B. [LS04b].

Wir möchten an dieser Stelle noch kurz darauf hinweisen, dass mittels XQuery 1.0 ausschließlich *lesende* Zugriffe auf XML-Daten möglich sind. XQuery erlaubt es also *nicht*, Änderungen am XML-Datenbestand vorzunehmen. Unter dem Namen *XQuery Update Facility* [W3C07f] wird vom W3C momentan jedoch eine Erweiterung von XQuery entwickelt, welche auch ein *änderndes* Arbeiten erlauben wird. Es handelt sich dabei allerdings (noch) nicht um eine offizielle W3C-Empfehlung, sondern lediglich um einen Entwurf *(W3C Working Draft)*. Eine ausführliche Betrachtung dieses Entwurfs erfolgt in [MB06].

2.10 XQuery-Datenmodell

Das im Jahr 2007 vom W3C als offizielle Empfehlung verabschiedete *XQuery 1.0 and XPath 2.0 Data Model* [W3C07e] — kurz *XQuery-Datenmodell* — bildet (wie sein Name nahelegt) die Grundlage für die Anfragesprachen XQuery 1.0 und XPath 2.0 (Abschnitt 2.9). Darüber hinaus basiert aber auch ein in der aktuellen SQL-Norm definierter SQL-Basisdatentyp auf dem XQuery-Datenmodell, sodass das XQuery-Datenmodell auch für die SQL-Norm von entscheidender Bedeutung ist. Wir werden hierauf im Abschnitt 2.11.3 näher eingehen.

Da eine umfassende Vorstellung des XQuery-Datenmodells [Kie05] den Rahmen dieser Arbeit sprengen würde, beschränken wir uns im Folgenden auf jene Aspekte, die für das Verständnis der vorliegenden Arbeit hilfreich sind. Für eine ausführlichere Beschreibung des XQuery-Datenmodells verweisen wir auf die Fachliteratur, z. B. [LS04b, MB06], bzw. die W3C-Spezifikation [W3C07e].

Das grundlegende Konstrukt des XQuery-Datenmodells ist die *Sequenz*. Eine Sequenz ist eine geordnete Folge, die aus keinem, einem oder mehreren *Sequenzeinträgen* besteht,

wobei es sich bei jedem Sequenzeintrag entweder um einen *atomaren Wert* oder um einen *Knoten* handelt. Bevor wir die Sequenzen im Abschnitt 2.10.3 detaillierter vorstellen, betrachten wir in den Abschnitten 2.10.1 und 2.10.2 zunächst die atomaren Werte sowie die Knoten.

2.10.1 Atomare Werte

Die Definition der atomaren Werte basiert auf so genannten *atomaren Typen (atomic types)*. Zu diesen atomaren Typen zählen einerseits bestimmte in XML-Schema vordefinierte (einfache) Datentypen wie *string*, *boolean* oder *time* (vgl. Abschnitt 2.7.3) sowie andererseits vom XQuery-Datenmodell neu eingeführte Datentypen wie *untypedAtomic* (Abschnitt 2.10.2.4). Darüber hinaus ist jeder Datentyp, der durch Einschränkung eines atomaren Typs gebildet wurde, selbst wieder ein atomarer Typ. Somit stellt beispielsweise auch unser im Abschnitt 2.7.3 definierter Typ *unserString* einen *atomaren* Typ dar, da dieser durch Einschränkung eines atomaren Typs *(string)* erzeugt wurde. Bei Datentypen, die durch Listenbildung oder Vereinigung entstanden sind, handelt es sich hingegen jedoch *nicht* um atomare Typen.

Ein *atomarer Wert (atomic value)* ist ein Wert eines atomaren Typs. Der atomare Wert ist dabei mit dem Typnamen des entsprechenden atomaren Typs gekennzeichnet. Wir können uns einen atomaren Wert damit als geordnetes Paar vorstellen, das aus dem eigentlichen Wert (z. B. *5*) und dem Typnamen (z. B. *integer*) besteht.

Strenggenommen werden Typnamen im XQuery-Datenmodell allerdings als Tripel repräsentiert und bestehen aus einem optionalen Namensraumpräfix, einem optionalen Namensraumnamen sowie dem (eigentlichen) Typnamen (vgl. Abschnitt 2.7.2). Für Datentypen, die in XML-Schema oder im XQuery-Datenmodell vordefiniert sind, lautet der Namensraumname *http://www.w3.org/2001/XMLSchema* und als Namensraumpräfix wird üblicherweise *xs* verwendet. Bei selbst definierten Datentypen (wie beispielsweise *unserString*) treten abweichende Namensraumnamen (z. B. *http://www.meineURL.de/Einkauf*) bzw. Namensraumpräfixe (z. B. *thm*) auf.

Ein atomarer Wert entspricht (strenggenommen) also einem geordneten Paar, bestehend aus dem eigentlichen Wert und dem durch ein Tripel repräsentierten Typnamen. *Abbildung 2.22* zeigt ein Beispiel für einen atomaren Wert des in XML-Schema vordefinierten (atomaren) Typs *integer*.

```
[5, [xs, http://www.w3.org/2001/XMLSchema, integer]]
```

Abbildung 2.22: Beispiel für einen atomaren Wert

Um eine bessere Lesbarkeit zu gewährleisten, werden wir im weiteren Verlauf dieser Arbeit auf die explizite Angabe der Namensraumpräfix- und Namensraumnamen-Bestandteile von Typnamen verzichten. Dies gilt nicht nur für Typnamen von atomaren Werten, sondern auch für Typnamen von Knoten (vgl. Abschnitt 2.10.2.4). Beispielsweise verwenden wir im Folgenden die Bezeichnung *integer* stellvertretend für den Typnamen *[xs, http://www.w3.org/2001/XMLSchema, integer]*. Wir möchten an dieser Stelle aber darauf hinweisen, dass es sich hierbei lediglich um eine Maßnahme zur Verbesserung der Lesbarkeit handelt und wir damit *keine* Einschränkung der Konzepte des XQuery-Datenmodells vornehmen.

2.10.2 Knoten

Im Abschnitt 2.10.2.1 stellen wir zunächst die vom XQuery-Datenmodell unterstützten Knotenarten vor. Möglichen Vater-Kind-Beziehungen zwischen Knoten widmet sich anschließend Abschnitt 2.10.2.2. Knoten besitzen gewisse Eigenschaften, zudem sind verschiedene Zugriffsfunktionen auf den Knoten definiert. Nähere Erläuterungen hierzu enthält Abschnitt 2.10.2.3. Knoten (bestimmter Knotenarten) können mit Typinformationen angereichert sein. Mit dieser Thematik beschäftigt sich Abschnitt 2.10.2.4. Zudem haben alle Knoten einen *textuellen Wert* und einen *getypten Wert*. Wir werden dies im Abschnitt 2.10.2.5 detaillierter betrachten.

2.10.2.1 Knotenarten

Wie auch XPath 1.0 (Abschnitt 2.9) unterscheidet das XQuery-Datenmodell zwischen sieben verschiedenen Knotenarten. Es handelt sich dabei um *Dokumentknoten*, *Elementknoten*, *Attributknoten*, *Namensraumknoten*, *Verarbeitungsanweisungsknoten*, *Kommentarknoten* und *Textknoten*. Diese Knotenarten ähneln den XPath-1.0-Knotenarten sehr stark [MB06]. Wir werden nun kurz auf die einzelnen Knotenarten eingehen.

Ein Dokumentknoten (*Document Node*, kurz *D-Knoten*) repräsentiert ein XML-Dokument. Der Begriff des XML-Dokuments wird vom XQuery-Datenmodell allerdings weiter gefasst als von der XML-Spezifikation. Während die XML-Spezifikation verlangt, dass ein XML-Dokument *genau ein* Wurzelelement enthält (vgl. Abschnitt 2.7.1), verzichtet das XQuery-Datenmodell auf diese Forderung. Somit ist nicht jedes XML-Dokument im Sinne des XQuery-Datenmodells auch ein wohlgeformtes XML-Dokument entsprechend der XML-Spezifikation.

Elementknoten (*Element Nodes*, kurz *E-Knoten*) und Attributknoten (*Attribute Nodes*, kurz *A-Knoten*) entsprechen Elementen bzw. Attributen aus XML 1.0. Wie beim XML-Infoset (Abschnitt 2.7.4) wird die Reihenfolge der einem Element zugeordneten Attribute allerdings als unerheblich erachtet. Laut XQuery-Datenmodell besitzen die zu einem Elementknoten gehörenden Attributknoten zwar eine stabile Reihenfolge — diese Reihenfolge ist aber implementierungsabhängig, also sozusagen „beliebig“.

Ein Namensraumknoten (*Namespace Node*, kurz *N-Knoten*) repräsentiert die Bindung eines Namensraumnamens an ein Namensraumpräfix. Die Reihenfolge der Namensraumbindungen eines Elements wird vom XQuery-Datenmodell dabei als unwesentlich angesehen. Die einem Elementknoten zugeordneten Namensraumknoten besitzen deshalb (analog zu den Attributknoten) zwar eine stabile, aber *beliebige* Reihenfolge. Bei den Namensraumknoten handelt es sich übrigens um eine stark umstrittene Knotenart, über deren Aufnahme ins XQuery-Datenmodell lange Zeit intensiv diskutiert wurde. Wir werden diese Thematik im Abschnitt 3.7 vertiefen.

Verarbeitungsanweisungsknoten (*Processing Instruction Nodes*, kurz *P-Knoten*), *Kommentarknoten* (*Comment Nodes*, kurz *C-Knoten*) und Textknoten (*Text Nodes*, kurz *T-Knoten*) stehen für XML-Verarbeitungsanweisungen, XML-Kommentare bzw. textuelle XML-Inhalte. Zusammenhängender Text (also Text, der weder durch Elemente, Verarbeitungsanweisungen noch Kommentare unterbrochen ist) wird im XQuery-Datenmodell stets durch einen *einzelnen* Textknoten repräsentiert — eine (unnötige) Aufsplittung in mehrere Textknoten ist explizit verboten.

2.10.2.2 Vater-Kind-Beziehungen zwischen Knoten

Knoten bestimmter Knotenarten können zueinander in einer Vater-Kind-Beziehung stehen. Das XQuery-Datenmodell erlaubt dabei ausschließlich D- und E-Knoten, Kindknoten zu besitzen. Als Kindknoten sind (in beiden Fällen) E-, P-, C- und T-Knoten zulässig. D-, A- und N-Knoten dürfen nicht als Kindknoten auftreten.

Einem E-Knoten (nicht aber einem D-Knoten) können A- und N-Knoten zugeordnet sein. Obwohl diese A- und N-Knoten dann nicht als Kinder des E-Knotens gelten, handelt es sich bei dem entsprechenden E-Knoten um den Vater dieser A- bzw. N-Knoten. Zwischen E-Knoten einerseits und A- bzw. N-Knoten andererseits sind also „einseitige“ Vater-Kind-Beziehungen möglich.

Die Zulässigkeit von Vater-Kind-Beziehungen zwischen Knoten verschiedener Knotenarten wird in *Abbildung 2.23* veranschaulicht. Ein Haken (✓) kennzeichnet dabei eine erlaubte „traditionelle“ (also beidseitige) Vater-Kind-Beziehung, während zulässige einseitige Vater-Kind-Beziehungen durch einen Stern (∗) symbolisiert werden. Ein Minuszeichen (−) drückt aus, dass weder eine einseitige noch eine beidseitige Vater-Kind-Beziehung möglich ist.

		Kindknoten						
		D	**E**	**A**	**N**	**P**	**C**	**T**
Vaterknoten	**D**	−	✓	−	−	✓	✓	✓
	E	−	✓	∗	∗	✓	✓	✓

Abbildung 2.23: Mögliche Vater-Kind-Beziehungen zwischen Knoten

Sowohl D- als auch E-Knoten dürfen zwar Kinder besitzen — sie müssen aber nicht. E-, A-, N-, P-, C- und T-Knoten können zwar einen Vater haben — sie brauchen aber nicht. Das XQuery-Datenmodell erlaubt für alle sieben Knotenarten kinder- und vaterlose Knoten. Damit sind u. a. beispielsweise „freistehende“ Attribute zulässig, die keinem Element zugeordnet sind. Dies ist ein wesentlicher Unterschied zu XML 1.0, wo Attribute nur in Verbindung mit Elementen auftreten konnten.

Besitzt ein D- oder E-Knoten mehrere T-Knoten als Kinder, so dürfen keine zwei dieser T-Kindknoten direkt aufeinanderfolgen. Zwei T-Kindknoten müssen also stets durch einen oder mehrere E-, P- oder C-Kindknoten getrennt sein. Dies ist eine Konsequenz aus der Forderung, zusammenhängenden Text stets durch einen *einzelnen* T-Knoten zu repräsentieren (vgl. Abschnitt 2.10.2.1).

Ein XML-Dokument im Sinne der XML-Spezifikation besitzt (wie bereits erwähnt) *genau ein* Wurzelelement. Unter den Kindern eines D-Knotens, der für ein solches wohlgeformtes XML-Dokument steht, muss sich folglich *genau ein* E-Knoten befinden. Dieser E-Kindknoten entspricht dann dem Wurzelelement. Das XQuery-Datenmodell erlaubt jedoch auch D-Knoten, die keine E-Kindknoten besitzen — zudem sind D-Knoten mit mehreren E-Kindknoten zulässig. Derartige (vom XQuery-Datenmodell unterstützte) D-Knoten entsprechen XML-Dokumenten, die keine bzw. mehrere Wurzelelemente enthalten und die damit nicht wohlgeformt sind. Hierin spiegelt sich wider, dass der Begriff des XML-Dokuments vom XQuery-Datenmodell weiter gefasst wird als von der XML-Spezifikation.

2.10.2.3 Knoteneigenschaften und Zugriffsfunktionen

Knoten besitzen (in Abhängigkeit von ihrer Knotenart) bestimmte *Eigenschaften (properties)*. Für E-Knoten sieht das XQuery-Datenmodell u. a. beispielsweise die Eigenschaften *node-name (Knotenname)* und *children (Kinder)* vor. Da C-Knoten (im Gegensatz zu E-Knoten) weder einen Namen haben noch Kinder besitzen können, existiert für sie keine der beiden zuvor genannten Eigenschaften. Stattdessen gibt es für C-Knoten die Eigenschaft *content (Inhalt)*. Auch für die restlichen Knotenarten definiert das XQuery-Datenmodell passende Eigenschaften. Eine umfassende Übersicht hierüber findet sich in [W3C07e].

In Ergänzung zu den Knoteneigenschaften führt das XQuery-Datenmodell diverse *Zugriffsfunktionen (accessors)* ein. Diese Zugriffsfunktionen lassen sich auf Knoten anwenden und liefern (basierend auf den Knoteneigenschaften) gewisse Informationen zurück. So existiert beispielsweise die Zugriffsfunktion *node-name*, die, angewendet auf einen Knoten, den Namen des Knotens zurückgibt (sofern dieser einen Namen besitzt). Ferner wird u. a. auch die Zugriffsfunktion *children* bereitgestellt, mit der sich die Kinder eines Knotens ermitteln lassen. Für eine detaillierte Beschreibung der angebotenen Zugriffsfunktionen verweisen wir auf [W3C07e].

Während die Knoteneigenschaften von der Knotenart abhängen, sind alle Zugriffsfunktionen für *alle* Knotenarten definiert. Wird eine Information abgefragt, die aufgrund der Knotenart generell nicht verfügbar ist, so liefert die Zugriffsfunktion als Ergebnis eine leere Sequenz zurück. Dies ist beispielsweise der Fall, wenn die Zugriffsfunktion *node-name* auf einen C-Knoten angewendet wird — C-Knoten sind stets namenlos.

Bei den Zugriffsfunktionen handelt es sich allerdings *nicht* um Funktionen, die dem Anwendungsprogramm bzw. dem Nutzer direkt zur Verfügung stehen. Stattdessen werden die Zugriffsfunktionen innerhalb der XQuery-Datenmodell-Spezifikation lediglich als Hilfsmittel genutzt, um die Informationen zu definieren, die *prinzipiell* zugreifbar sein sollen.

Zwischen den Knoteneigenschaften (durch welche festgelegt ist, welche Informationen existieren) und den Zugriffsfunktionen (durch welche definiert ist, welche Informationen zugreifbar sind) besteht naheliegenderweise ein enger Zusammenhang. In vielen Fällen gibt die Zugriffsfunktion einfach den Inhalt der passenden Knoteneigenschaft zurück. So liefert z. B. die Zugriffsfunktion *node-name*, angewendet auf einen E- oder A-Knoten, den Wert der gleichnamigen Knoteneigenschaft.

In einigen Situationen gibt es zwischen den Knoteneigenschaften und den Zugriffsfunktionen jedoch Unterschiede. Wir betrachten hierzu die P-Knoten, welche zwar u. a. die Knoteneigenschaft *target (Verarbeitungsanweisungsziel)* besitzen, aber keine Knoteneigenschaft *node-name*. Dennoch liefert die Zugriffsfunktion *node-name*, angewendet auf einen P-Knoten, einen Wert zurück — nämlich den Inhalt der *target*-Eigenschaft. Während P-Knoten im Sinne der Knoteneigenschaften also namenlos sind, besitzen sie entsprechend der Zugriffsfunktionen einen Namen — das Verarbeitungsanweisungsziel wird dabei als Knotenname aufgefasst. Sofern zwischen Knoteneigenschaften und Zugriffsfunktionen derartige Differenzen bestehen, orientieren wir uns im Folgenden an den Zugriffsfunktionen.

E-, A- und P-Knoten haben einen Namen. Strenggenommen handelt es sich bei einem Knotennamen (analog zu einem Typnamen) um ein Tripel, bestehend aus einem optionalen Namensraumpräfix, einem optionalen Namensraumnamen sowie dem (eigentlichen) Knotennamen. Ein *Artikel*-Element aus Abschnitt 2.7.1 könnte im XQuery-Datenmodell beispielsweise durch einen E-Knoten repräsentiert werden, dessen Knotenname sich aus

dem Namensraumpräfix *thm*, dem Namensraumnamen *http://www.meineURL.de/Einkauf* und dem lokalen Namensbestandteil *Artikel* zusammensetzt. Wie bei Typnamen werden wir im weiteren Verlauf dieser Arbeit auf die explizite Angabe der Namensraumpräfix- und Namensraumnamen-Bestandteile verzichten. Diese Maßnahme dient jedoch lediglich zur Verbesserung der Lesbarkeit und bedeutet keine Einschränkung der Konzepte des XQuery-Datenmodells.

2.10.2.4 Typannotationen

Ein wohlgeformtes XML-Dokument kann gegen ein XML-Schema validiert werden, wobei geprüft wird, ob das XML-Dokument den Anforderungen des XML-Schemas genügt (vgl. Abschnitt 2.7.3). Bei einer solchen Validierung wird insbesondere getestet, ob die Elemente und Attribute die für sie vorgesehenen Datentypen haben. Würde unser XML-Dokument aus Abschnitt 2.7.1 beispielsweise gegen das in Abschnitt 2.7.3 angedeutete XML-Schema validiert, so würde u. a. ermittelt, ob die *Artikel*-Elemente den Typ *ArtikelTyp* und die *Anzahl*-Attribute den Typ *integer* besitzen. Da unser XML-Dokument alle Anforderungen des XML-Schemas erfüllt, verliefe die Validierung erfolgreich.

Werden Elemente und Attribute, deren Datentypen (z. B. in Folge einer Validierung) bekannt sind, im XQuery-Datenmodell durch E- bzw. A-Knoten repräsentiert, so besitzen diese Knoten den jeweiligen Datentyp als *Typnamen (type-name)*. Ein *Artikel*-Element entspräche also einem E-Knoten mit Typnamen *ArtikelTyp*, während ein *Anzahl*-Attribut durch einen A-Knoten mit Typnamen *integer* dargestellt würde.

E- und A-Knoten besitzen aber auch dann einen Typnamen, falls für sie keine Typinformationen vorliegen. Dies ist beispielsweise der Fall, wenn Elemente und Attribute eines nicht-validierten XML-Dokuments repräsentiert werden. Die E-Knoten erhalten dann den Typnamen *untyped*, die A-Knoten den Typnamen *untypedAtomic*.

Sowohl der komplexe Datentyp *untyped* als auch der einfache (und zugleich atomare) Datentyp *untypedAtomic* signalisieren, dass keine Typinformationen vorliegen. Beide Datentypen wurden vom XQuery-Datenmodell neu eingeführt. Es waren zwei verschiedene Datentypen erforderlich, da für E-Knoten (die möglicherweise zugeordnete A-Knoten sowie E-Kindknoten haben können) ein komplexer Typ benötigt wird, während A-Knoten stets einen einfachen Typ besitzen müssen (vgl. Abschnitt 2.7.3).

Außer den zuvor betrachteten Möglichkeiten einer erfolgreichen Validierung und einer nicht erfolgten Validierung kann es jedoch auch sein, dass zwar eine Validierung versucht wurde, diese aber scheiterte, weil z. B. ein Element oder Attribut nicht den Schemavorgaben genügte. Bei einer Repräsentation im XQuery-Datenmodell besitzt ein entsprechender E-Knoten dann den Typnamen *anyType* und ein A-Knoten den Typnamen *anySimpleType*.

Der komplexe Datentyp *anyType* und der einfache Datentyp *anySimpleType* stammen aus XML-Schema und wurden vom XQuery-Datenmodell übernommen. Ähnlich wie *untyped* und *untypedAtomic* dienen diese Datentypen zur Kennzeichnung von E- bzw. A-Knoten, für die keine (genaueren) Typinformationen verfügbar sind. Zwischen den für E-Knoten genutzten Datentypen *untyped* und *anyType* gibt es jedoch einen wesentlichen Unterschied, auf den wir im Folgenden näher eingehen.

Hat ein E-Knoten den Typnamen *untyped*, so wird erzwungen, dass alle seine E-Nachfahrenknoten (also E-Kindknoten, E-Enkelknoten, E-Urenkelknoten usw.) ebenfalls

den Typnamen *untyped* besitzen. Darüber hinaus müssen alle dem E-Knoten zugeordneten A-Knoten vom Typ *untypedAtomic* sein. Der Typname *untyped* signalisiert also nicht nur, dass der E-Knoten selbst ohne Typinformation ist, sondern auch, dass alle seine E-Nachfahrenknoten sowie alle ihm zugeordneten A-Knoten ohne Typinformation sind. Im Gegensatz hierzu gibt es bezüglich des Datentyps *anyType* keine derartige Forderung. Ein E-Knoten mit Typnamen *anyType* darf also durchaus zugeordnete A-Knoten sowie E-Kindknoten mit vorliegenden (konkreten) Typinformationen besitzen.

Bisher haben wir beschrieben, welche Typnamen E- und A-Knoten besitzen, wenn diese aus Elementen bzw. Attributen eines XML-Dokuments hergeleitet werden. Dabei hatten wir unterschieden, ob das XML-Dokument erfolgreich validiert wurde, ob keine Validierung durchgeführt wurde oder ob die Validierung gescheitert ist. Obwohl E- und A-Knoten (theoretisch) zwar stets Elemente und Attribute eines XML-Dokuments repräsentieren (vgl. Abschnitt 2.10.2.1), bedeutet dies aber nicht, dass ein entsprechendes XML-Dokument auch tatsächlich existieren muss. Alle Knoten (einschließlich E- und A-Knoten) sind auch *direkt* (also ohne vorliegendes XML-Dokument) konstruierbar. Dabei muss lediglich sichergestellt sein, dass alle Forderungen des XQuery-Datenmodells erfüllt sind. Hat beispielsweise ein direkt konstruierter E-Knoten den Typnamen *untyped*, so müssen alle ihm zugeordneten A-Knoten den Typnamen *untypedAtomic* und alle E-Nachfahrenknoten den Typnamen *untyped* besitzen.

Wir möchten nun noch kurz eine Besonderheit der E-Knoten betrachten — die so genannte *nilled*-Eigenschaft. Diese Knoteneigenschaft hat im Normalfall den Wert *false*. Besitzt diese Eigenschaft jedoch den Wert *true*, so wird dadurch ausgedrückt, dass ein E-Knoten ohne E- und T-Kindknoten selbst dann als gültig (d. h. typkonform) angesehen werden soll, wenn der entsprechende Datentyp für ihn eigentlich E- bzw. T-Kindknoten fordert. Mittels dieser Knoteneigenschaft können also gewisse Forderungen der Typdefinition außer Kraft gesetzt werden. Für E-Knoten mit Typnamen *untyped* sowie für E-Knoten mit E- und/oder T-Kindknoten darf die *nilled*-Eigenschaft allerdings nicht den Wert *true* annehmen. Der Wert der *nilled*-Eigenschaft ist bei der Bestimmung des getypten Werts relevant (vgl. Abschnitt 2.10.2.5).

Außer E- und A-Knoten besitzen auch T-Knoten einen Typnamen. Der Typname eines T-Knotens lautet jedoch stets *untypedAtomic* und stellt somit keine „echte“ Typinformation dar. Die Knoten der restlichen vier Knotenarten (also D-, N-, P- und C-Knoten) besitzen hingegen keinen Typnamen.

Wir werden im weiteren Verlauf dieser Arbeit den Typnamen eines Knotens als dessen Typ auffassen. Beispielsweise verstehen wir unter einem „E-Knoten des Typs *ArtikelTyp*“ einen E-Knoten mit dem entsprechenden Typnamen. Desweiteren unterscheiden wir konsequent zwischen der *Art* (z. B. *E* bzw. *Element*), dem *Namen* (z. B. *Artikel*) und dem *Typ* (z. B. *ArtikelTyp*) eines Knotens.

2.10.2.5 Textueller und getypter Wert

Jeder Knoten hat entsprechend dem XQuery-Datenmodell sowohl einen *textuellen Wert (string value)* als auch einen *getypten Wert (typed value)*. Wir möchten dies kurz anhand eines kleinen Beispiels veranschaulichen und betrachten hierzu einen A-Knoten des Typs *integer*, der das Attribut *Anzahl="2"* unseres XML-Dokuments aus Abschnitt 2.7.1 repräsentiere. Dieser A-Knoten besitzt als textuellen Wert die Zeichenkette *"2"* und als

getypten Wert den *Integer*-Wert *2*. Ein wesentlicher Unterschied zwischen dem textuellen und dem getypten Wert dieses A-Knotens besteht darin, dass letzterer innerhalb von arithmetischen Operationen genutzt werden kann — z. B. ließe sich die Gesamtzahl der gekauften Artikel durch Aufsummierung der getypten Werte aller A-Knoten namens *Anzahl* ermitteln. Der textuelle Wert, der vom Typ *string* ist, kann hingegen nicht in derartigen Arithmetikoperationen verwendet werden.

Da die Definition der getypten Werte auf den textuellen Werten basiert, werden wir zunächst näher auf die textuellen Werte eingehen. Im Anschluss betrachten wir dann die getypten Werte. Diese wiederum bilden die Grundlage für die von uns entwickelte *typed-value-orientierte Repräsentation* von Sequenzen, die wir im Kapitel 3 vorstellen.

Der textuelle Wert eines Knotens besitzt stets den Typ *string*. Für D- und E-Knoten wird er durch Verkettung der Inhalte aller T-Nachfahrenknoten gebildet, wobei die Verkettung entsprechend der Präorder-Reihenfolge der betroffenen T-Knoten erfolgt. Falls ein D- bzw. E-Knoten keine T-Nachfahrenknoten hat, so handelt es sich bei seinem textuellen Wert um die leere Zeichenkette, also die Zeichenkette der Länge 0. Der textuelle Wert eines A-Knotens entspricht dem (in eine Zeichenkette umgewandelten) Wert des Attributs, welches durch den A-Knoten repräsentiert wird. Der textuelle Wert eines N-Knotens ist der Namensraumname, der durch den N-Knoten an ein Namensraumpräfix gebunden wird. Als textueller Wert eines P-, C- oder T-Knotens gilt deren Inhalt. Der Zusammenhang zwischen Knotenart und textuellem Wert wird in *Abbildung 2.24* nochmals überblicksweise dargestellt.

Der getypte Wert eines N-, P- oder C-Knotens ist mit dem textuellen Wert identisch. Für D- und T-Knoten ergibt sich der getypte Wert durch Umwandlung des textuellen Werts in eine Instanz des Datentyps *untypedAtomic*. Bei A- und E-Knoten wird der getypte Wert in Abhängigkeit des Knotentyps aus dem textuellen Wert ermittelt, wobei bei E-Knoten auch die *nilled*-Eigenschaft eine Rolle spielt (vgl. Abschnitt 2.10.2.4). Für E-Knoten mit einem komplexen Typ ist auch die jeweilige Unterart des komplexen Typs (z. B. komplexer Typ mit *einfachem Inhalt*) von Bedeutung (vgl. Abschnitt 2.7.3). Abbildung 2.24 fasst die von der Knotenart abhängige Bestimmung des getypten Werts nochmals kurz zusammen.

<table>
<tr><th>textueller Wert</th><th>Knotenart</th><th>getypter Wert</th></tr>
<tr><td rowspan="2">Verkettung der Inhalte der T-Nachfahrenknoten in Präorder-Reihenfolge</td><td>D</td><td>textueller Wert als untypedAtomic</td></tr>
<tr><td>E</td><td>Ermittlung aus textuellem Wert in Abhängigkeit vom Typ und von der nilled-Eigenschaft</td></tr>
<tr><td>Wert des repräsentierten Attributs als string</td><td>A</td><td>Ermittlung aus textuellem Wert in Abhängigkeit vom Typ</td></tr>
<tr><td>Namensraumname</td><td>N</td><td rowspan="3">identisch mit textuellem Wert</td></tr>
<tr><td rowspan="3">Knoteninhalt</td><td>P</td></tr>
<tr><td>C</td></tr>
<tr><td>T</td><td>textueller Wert als untypedAtomic</td></tr>
</table>

Abbildung 2.24: Textueller und getypter Wert eines Knotens

Der getypte Wert eines A-Knotens vom Typ *anySimpleType* oder *untypedAtomic* entspricht dem textuellen Wert, umgewandelt nach *untypedAtomic*. Gleiches gilt für E-Knoten mit

nilled-Eigenschaft *false*, die einen komplexen Typ mit gemischtem Inhalt besitzen oder vom Typ *untyped*, *anySimpleType* bzw. *anyType* sind. Unabhängig von der *nilled*-Eigenschaft besitzen E-Knoten eines komplexen Typs mit leerem Inhalt die leere Folge als getypten Wert. Die leere Folge als getypter Wert tritt auch bei allen E-Knoten mit *nilled*-Eigenschaft *true* auf, die *nicht* von einem komplexen Typ mit Nur-Element-Inhalt sind. Für E-Knoten mit einem komplexen Typ mit Nur-Element-Inhalt ist der getypte Wert (unabhängig von der *nilled*-Eigenschaft) nicht definiert.

Für A-Knoten, die *nicht* vom Typ *anySimpleType* oder *untypedAtomic* sind, sowie für E-Knoten mit *nilled*-Eigenschaft *false*, die einen einfachen Typ oder einen komplexen Typ mit einfachem Inhalt besitzen, handelt es sich beim getypten Wert um eine Folge aus keinem, einem oder mehreren atomaren Werten, möglicherweise unterschiedlicher Datentypen. Wir werden hierauf im Folgenden näher eingehen. Die *Abbildungen 2.25* und *2.26* fassen die Ermittlung des getypten Werts von E- bzw. A-Knoten zusammen. Wie aus Abbildung 2.25 ersichtlich (und in Abschnitt 2.10.2.4 bereits erwähnt), kann die *nilled*-Eigenschaft für E-Knoten des Typs *untyped* nicht den Wert *true* annehmen.

<table>
<tr><th>nilled = false</th><th>Typ des E-Knotens</th><th>nilled = true</th></tr>
<tr><td rowspan="4">textueller Wert
als untypedAtomic</td><td>untyped</td><td>(nicht möglich)</td></tr>
<tr><td>anySimpleType</td><td rowspan="5">leere Folge</td></tr>
<tr><td>anyType</td></tr>
<tr><td>komplexer Typ mit gemischtem Inhalt</td></tr>
<tr><td rowspan="2">Folge aus keinem, einem oder
mehreren atomaren Werten</td><td>einfacher Typ</td></tr>
<tr><td>komplexer Typ mit einfachem Inhalt</td></tr>
<tr><td>leere Folge</td><td>komplexer Typ mit leerem Inhalt</td></tr>
<tr><td>nicht definiert</td><td>komplexer Typ mit Nur-Element-Inhalt</td><td>nicht definiert</td></tr>
</table>

Abbildung 2.25: Getypter Wert eines E-Knotens

<table>
<tr><th>Typ des A-Knotens</th><th>getypter Wert</th></tr>
<tr><td>anySimpleType</td><td rowspan="2">textueller Wert als untypedAtomic</td></tr>
<tr><td>untypedAtomic</td></tr>
<tr><td>sonstiger Typ</td><td>Folge aus keinem, einem oder mehreren atomaren Werten</td></tr>
</table>

Abbildung 2.26: Getypter Wert eines A-Knotens

Für E-Knoten mit *nilled*-Eigenschaft *false* und einem einfachen Typ bzw. einem komplexen Typ mit einfachem Inhalt geschieht die Ermittlung des getypten Werts auf eine sehr naheliegende und intuitive Weise. Gleiches gilt für A-Knoten mit einem von *anySimpleType* und *untypedAtomic* abweichenden Typ. Wir werden die Bestimmung des getypten Werts anhand einiger Beispiele aufzeigen und verweisen für weitere Details auf die W3C-Spezifikationen [W3C07e] und [W3C07d].

Die drei folgenden Beispiele orientieren sich an unserem XML-Dokument aus Abschnitt 2.7.1 sowie dem im Abschnitt 2.7.3 angedeuteten XML-Schema.

- Ein A-Knoten besitze den Typ *integer* und den textuellen Wert *"1"*. Der getypte Wert des A-Knotens lautet dann *[1, integer]*, also *1* als Instanz des Typs *integer*.
- Ein E-Knoten mit einfachem (und zugleich atomarem) Typ *unserString* und dem textuellen Wert *"Kerze"* besitzt den getypten Wert *["Kerze", unserString]*.
- Ein E-Knoten sei vom Typ *PreisTyp*, wobei es sich um einen komplexen Typ mit einfachem Inhalt handelt, der als Erweiterung des Typs *decimal* definiert ist. Der E-Knoten habe den textuellen Wert *"21.05"*. Sein getypter Wert lautet damit *[21.05, decimal]*.

Im Folgenden betrachten wir noch kurz vier Beispiele, bei denen Datentypen, die durch Vereinigung und/oder Listenbildung gebildet wurden, involviert sind. Wir unterstellen hierbei die Existenz eines einfachen und atomaren Typs *JahresTyp*, der (als Einschränkung des Typs *integer*) natürliche Zahlen zwischen 1900 und 2100 zulasse.

- Ein A-Knoten besitze einen Listentyp, der basierend auf dem Typ *JahresTyp* definiert sei. Der textuelle Wert laute *"1976 1977 2006"*. Beim getypten Wert handelt es sich damit um die Folge *([1976, JahresTyp], [1977, JahresTyp], [2006, JahresTyp])* — also eine Folge, bestehend aus drei atomaren Werten des Typs *JahresTyp*. Bei der Ermittlung des getypten Werts wurden die Leerzeichen innerhalb des textuellen Werts (wie vom XQuery-Datenmodell vorgesehen) als Trennzeichen aufgefasst.
- Ein A-Knoten besitze (erneut) den Listentyp des vorigen Beispiels. Sein textueller Wert sei diesmal (eine leere Liste von Jahreszahlen repräsentierend) jedoch die leere Zeichenkette *("")*. Beim getypten Wert handelt es sich damit um eine leere Folge.
- Ein E-Knoten sei von einem Vereinigungstyp, der Werte der Datentypen *JahresTyp* und *boolean* zulasse. Als textuellen Wert besitze der Knoten die Zeichenkette *"true"*. Der getypte Wert lautet somit *[true, boolean]*.
- Ein E-Knoten sei von einem komplexen Typ mit einfachem Inhalt. Dieser komplexe Typ sei als Erweiterung eines Listentyps definiert, welcher wiederum auf dem Vereinigungstyp des vorigen Beispiels basiere. Der textuelle Wert laute *"true 2008 false"*. Beim getypten Wert handelt es sich damit um die Folge *([true, boolean], [2008, JahresTyp], [false, boolean])* — also eine Folge, bestehend aus drei atomaren Werten der Datentypen *boolean* bzw. *Jahrestyp*.

Beim getypten Wert eines E- oder A-Knotens kann es sich also um eine Folge *mehrerer* atomarer Werte (möglicherweise unterschiedlicher Datentypen) handeln. Listen- und Vereinigungstypen sind dabei nicht als Datentypen der einzelnen atomaren Werte zulässig, da es sich bei ihnen nicht um atomare Typen handelt (vgl. Abschnitt 2.10.1).

2.10.3 Sequenzen

Wie zu Beginn des Abschnitts 2.10 bereits erwähnt, handelt es sich bei Sequenzen um das zentrale Konstrukt des XQuery-Datenmodells. Sequenzen sind geordnete Folgen, die sich aus keinem, einem oder mehreren Sequenzeinträgen *(items)* zusammensetzen. Jeder Sequenzeintrag ist entweder ein atomarer Wert (Abschnitt 2.10.1) oder ein Knoten (Abschnitt 2.10.2).

Neben Sequenzen, die ausschließlich aus atomaren Werten bestehen, und Sequenzen, die nur Knoten enthalten, sind auch Sequenzen zulässig, in denen sowohl atomare Werte als auch Knoten vorkommen. Sequenzen dürfen Duplikate (also mehrfach vorkommende Sequenzeinträge) enthalten. Ein Ineinanderschachteln von Sequenzen ist *nicht* erlaubt.

Das XQuery-Datenmodell unterscheidet nicht zwischen einem Sequenzeintrag und einer Sequenz, die genau diesen einen Sequenzeintrag enthält. Eine Sequenz, die keinen einzigen Sequenzeintrag beinhaltet, wird als *leere Sequenz* bezeichnet.

Ein in der Sequenz auftretender Knoten kann (abhängig von seiner Knotenart) Kindknoten und/oder zugeordnete A- bzw. N-Knoten besitzen (vgl. Abschnitt 2.10.2.2). Der entsprechende Sequenzeintrag kann damit als Baum aufgefasst werden. Auch ein Sequenzeintrag, bei dem es sich um einen Knoten ohne Kinder und ohne zugeordnete A- bzw. N-Knoten handelt, lässt sich als Baum auffassen — nämlich als Baum, der nur aus diesem einen Knoten besteht. Somit entspricht eine Sequenz einer (möglicherweise leeren) geordneten Folge aus Bäumen und/oder atomaren Werten. Die Bäume sind dabei aus den Knoten der im Abschnitt 2.10.2.1 vorgestellten Knotenarten aufgebaut. Jeder Knoten besitzt eine eindeutige Identität, d. h., er ist identisch mit sich selbst, aber nicht identisch mit irgendeinem anderen Knoten.

Abbildung 2.27 zeigt eine Sequenz, die aus zwei atomaren Werten und vier Bäumen — also insgesamt sechs Sequenzeinträgen — besteht, wobei der zweite der vier Bäume lediglich einen Knoten umfasst. Die Buchstaben *D, E, A, N, P, C* bzw. *T* stehen für die Knotenart des jeweiligen Knotens. Atomare Werte sind durch den Buchstaben *V* (für *atomic value*) gekennzeichnet. Die einzelnen Sequenzeinträge sind (entsprechend ihrer Reihenfolge) durch Pfeile verbunden. Innerhalb der Bäume repräsentieren durchgehende Kanten *beidseitige* Vater-Kind-Beziehungen, während *einseitige* Vater-Kind-Beziehungen (vgl. Abschnitt 2.10.2.2) durch gestrichelte Kanten dargestellt sind. Wie im Abschnitt 2.10.2.1 erwähnt, ist die Reihenfolge der einem E-Knoten zugeordneten N- bzw. A-Knoten zwar nicht relevant, aber dennoch stabil. Der vierte Sequenzeintrag der abgebildeten Sequenz entspricht (in seiner Struktur) übrigens unserem XML-Dokument aus Abschnitt 2.7.1.

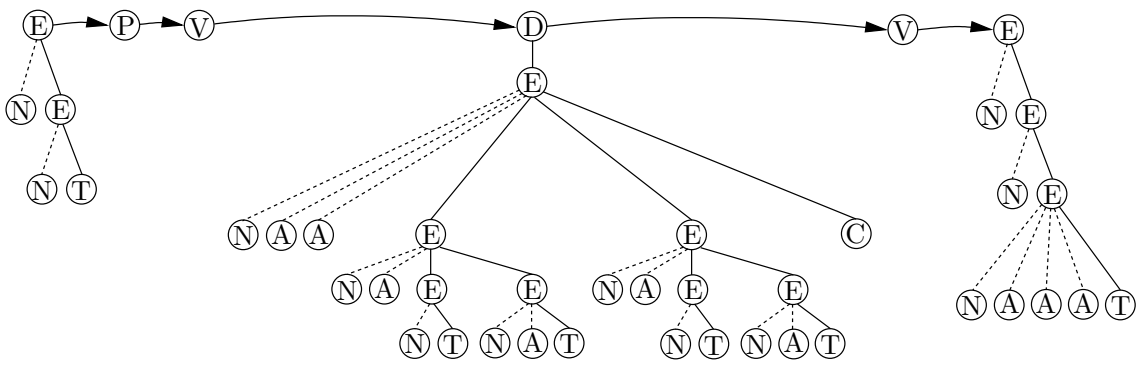


Abbildung 2.27: Beispiel für eine Sequenz

2.11 XML-Unterstützung durch die SQL-Norm

Bisher haben wir SQL-bezogene Themen (Abschnitte 2.1 bis 2.5) und XML-nahe Konzepte (Abschnitte 2.6 bis 2.10) separat betrachtet. In der Praxis besteht jedoch oftmals der Wunsch, „traditionelle“ SQL-Daten und XML-Daten *gemeinsam* zu verwalten und *integriert* auszuwerten [MB06]. Die sich daraus ergebende Notwendigkeit eines „Brückenschlags“ zwischen „SQL-Welt“ und „XML-Welt“ wurde von den SQL-Normungsgremien erkannt [EM01, EM02, Mül04]. Mit der offiziellen Verabschiedung von SQL:2003 [EMK+04, Tür03] erfolgte eine Erweiterung von SQL um XML-Funktionalität. Der hierzu eigens neu eingeführte Normteil *SQL/XML:2003* [ISO03b] enthielt dabei alle Reglungen, die im Zusammenhang mit der XML-Unterstützung Eingang in die SQL-Norm fanden. Im Jahr 2006 wurde dieser Normteil aber bereits durch eine überarbeitete Fassung mit der Bezeichnung *SQL/XML:2006* ersetzt. SQL/XML:2006 [ISO06, ISO07] ist damit offizieller Bestandteil der (derzeit aktuellen) SQL-Version SQL:2003.

Wir werden im Folgenden betrachten, wie XML-Werte (also XML-Dokumente bzw. Sequenzen) von der SQL-Norm unterstützt werden. Bevor wir in den Abschnitten 2.11.2 und 2.11.3 auf SQL/XML:2003 bzw. SQL/XML:2006 eingehen, widmet sich Abschnitt 2.11.1 zunächst der Situation, die vor der Veröffentlichung von SQL:2003 vorlag.

2.11.1 XML-Werte vor SQL:2003

Bevor SQL:2003 verabschiedet wurde, gab es in der SQL-Norm keinerlei XML-Unterstützung. Sollten XML-Dokumente in SQL-Tabellen gespeichert werden, musste der Nutzer bzw. das Anwendungsprogramm — als eine Art „Notbehelf“ — stets auf eine mit diversen Nachteilen behaftete (und nicht von der SQL-Norm erfasste) XML-Speicherungstechnik zurückgreifen [KM02].

Ein XML-Dokument konnte beispielsweise als einfache Zeichenkette in einer `VARCHAR`- oder `CLOB`-Spalte abgelegt werden, wobei es allerdings nicht auf seine Wohlgeformtheit überprüfbar war. Eine andere Möglichkeit bestand darin, das XML-Dokument in seine Bestandteile zu zerlegen und diese dann in einer oder mehreren Tabellen zu speichern. Bei diesem (als *Shredding* bezeichneten) Vorgehen war eine spätere Rekonstruktion des ursprünglichen XML-Dokuments allerdings mit einem enormen Aufwand verbunden [MB06].

Da eine umfassende Betrachtung sämtlicher XML-Speicherungstechniken hier zu weit führen würde, möchten wir für eine ausführliche Klassifikation, Gegenüberstellung und Bewertung der verschiedenen Verfahren auf die einschlägige Fachliteratur, z. B. [KM02], verweisen.

2.11.2 XML-Werte in SQL/XML:2003

SQL/XML:2003 (bis zur Ablösung durch SQL/XML:2006 Teil von SQL:2003) hat den auf dem XML-Infoset (Abschnitt 2.7.4) basierenden SQL-Basisdatentyp *XML* eingeführt. Damit wurde es erstmals auch aus Sicht der SQL-Norm möglich, Tabellen anzulegen, die in einer oder mehreren Spalten XML-Dokumente enthalten. Diese Normerweiterung war ein erster und zugleich grundlegender Schritt, um einen SQL-basierten integrierten Zugriff auf XML-Daten und „traditionelle“ SQL-Daten zu ermöglichen. Da nun ein „vollwertiger“

SQL-Datentyp `XML` zur Verfügung steht, ist (seitens des Nutzers bzw. des Anwendungsprogramms) ein Rückgriff auf die oben genannten XML-Speicherungstechniken nicht mehr nötig.

Bei einem XML-Wert (d. h. einer Instanz des SQL/XML:2003-Datentyps `XML`) kann es sich um den (SQL-)Nullwert oder um ein wohlgeformtes XML-Dokument handeln. Als zulässige XML-Werte im Sinne von SQL/XML:2003 gelten darüber hinaus auch (nicht wohlgeformte) XML-Dokumente, welche die Forderung verletzen, *genau ein* Wurzelelement zu besitzen. Zum Test, ob ein XML-Wert ein *wohlgeformtes* XML-Dokument ist, stellt SQL/XML:2003 das Prädikat `IS DOCUMENT` bereit.

2.11.3 XML-Werte in SQL/XML:2006

In SQL/XML:2006 (seit 2006 Teil von SQL:2003) wurde die XML-Unterstützung gegenüber SQL/XML:2003 deutlich ausgebaut. Mittels der neu eingeführten SQL-Funktion `XMLQUERY` ist es nun sogar möglich, XQuery-Anfragen (Abschnitt 2.9) innerhalb von SQL zu formulieren und auszuwerten [MB06, Zem04b]. Die wesentliche Neuerung von SQL/XML:2006 liegt aber im Wechsel des dem SQL-Basisdatentyp `XML` zugrunde gelegten Datenmodells [EM04a, Zem04a]. Im Gegensatz zu SQL/XML:2003 beruht dieser Datentyp nicht mehr auf einem XML-Infoset-basierten Datenmodell, sondern auf dem XQuery-Datenmodell (Abschnitt 2.10). Als XML-Werte sind damit beliebige (XQuery-)Sequenzen (Abschnitt 2.10.3) erlaubt. Dies schließt insbesondere Sequenzen ein, die aus mehreren Sequenzeinträgen bestehen. Ebenso kann es sich um Sequenzen handeln, deren Knoten mit Typinformationen angereichert sind (vgl. Abschnitt 2.10.2.4).

SQL/XML:2006 führt drei Untertypen des SQL-Datentyps `XML` ein, mit deren Hilfe sich die zulässigen XML-Werte (bei Bedarf) einschränken lassen. Es handelt sich dabei um `XML(SEQUENCE)`, `XML(CONTENT)` und `XML(DOCUMENT)` [MB06]. Alle drei Untertypen erlauben das Auftreten von SQL-Nullwerten. Wir werden dies bei den folgenden Beschreibungen als bekannt voraussetzen und die Zulässigkeit der Nullwerte nicht jedes Mal explizit erwähnen.

Eine Instanz des SQL-Datentyps `XML(SEQUENCE)` ist eine beliebige Sequenz, also eine beliebige geordnete Folge aus Bäumen und/oder atomaren Werten (vgl. Abschnitt 2.10.3). Der Datentyp `XML(CONTENT)` erlaubt hingegen nur einen einzelnen Baum, der als Wurzel einen D-Knoten haben muss. Es wird dabei allerdings *nicht* gefordert, dass unter den Kindern dieses D-Knotens genau ein E-Knoten vorzukommen hat. Werte dieses Datentyps entsprechen somit XML-Dokumenten im Sinne des XQuery-Datenmodells (vgl. Abschnitt 2.10.2.1). Zur Repräsentation von wohlgeformten XML-Dokumenten (also XML-Dokumenten im Sinne der XML-Spezifikation) dient der Untertyp `XML(DOCUMENT)`. Ein Wert dieses Datentyps ist — wie bei `XML(CONTENT)` — ein einzelner Baum mit einem D-Knoten als Wurzel. Unter den Kindern dieses D-Knotens muss sich — im Gegensatz zu `XML(CONTENT)` — aber *genau ein* E-Knoten befinden.

Die drei vorgestellten Untertypen des Datentyps `XML` sind nicht disjunkt. Jede Instanz von `XML(DOCUMENT)` ist stets auch eine Instanz von `XML(CONTENT)` und jede Instanz von `XML(CONTENT)` ist stets auch eine Instanz von `XML(SEQUENCE)`. Zum Test, ob es sich bei einem XML-Wert um ein XML-Dokument im Sinne des XQuery-Datenmodells oder um ein wohlgeformtes XML-Dokument im Sinne der XML-Spezifikation handelt, stellt SQL/XML:2006 die Prädikate `IS CONTENT` bzw. `IS DOCUMENT` bereit.

Für `XML(DOCUMENT)` und `XML(CONTENT)` sieht die SQL-Norm eine weitere Verfeinerung in Unter(unter)typen vor. Beispielsweise wird für Werte vom SQL-Datentyp `XML(DOCUMENT(UNTYPED))` verlangt, dass jeder im Baum enthaltene E-Knoten vom Typ *untyped* und jeder A-Knoten vom Typ *untypedAtomic* sein muss. Instanzen dieses SQL-Datentyps entsprechen also nicht-validierten XML-Dokumenten (vgl. Abschnitt 2.10.2.4). Um ausschließlich XML-Dokumente zuzulassen, die bezüglich eines bestimmten XML-Schemas gültig sind, kann der Datentyp `XML(DOCUMENT(XMLSCHEMA))` genutzt werden. `XML(DOCUMENT(ANY))` erlaubt hingegen beliebige wohlgeformte XML-Dokumente, also sowohl nicht-validierte als auch gültige. Für `XML(CONTENT)` — nicht jedoch für `XML(SEQUENCE)` — existiert eine analoge Verfeinerung in Unter(unter)typen.

Mittels der in SQL/XML:2006 neu eingeführten Funktion `XMLVALIDATE` ist es möglich, eine Sequenz gegen ein XML-Schema zu validieren. Die zu validierende Sequenz darf dabei aus *mehreren* Bäumen (also *mehreren* Sequenzeinträgen) bestehen. Bei der Validierung wird für jeden einzelnen Baum geprüft, ob er den Anforderungen des XML-Schemas genügt. Verlaufen all diese Prüfungen erfolgreich, so ist die Sequenz bezüglich des XML-Schemas *gültig* und die einzelnen Knoten werden mit entsprechenden Typinformationen angereichert (vgl. Abschnitt 2.10.2.4). Bei der Validierung einer Sequenz wird also nicht nur überprüft, ob alle Anforderungen des XML-Schemas erfüllt sind, sondern es findet (eine *erfolgreiche* Validierung vorausgesetzt) auch eine Anreicherung mit Typinformationen statt. Dies bedeutet, dass sich die Typen — und somit auch die getypten Werte — der Knoten im Zuge der Validierung ändern können (vgl. Abschnitt 2.10.2.5).

SQL/XML:2006 macht einige (teilweise willkürlich erscheinende) Einschränkungen bezüglich der mittels `XMLVALIDATE` validierbaren Sequenzen. Beispielsweise sind in der Wurzelebene einer zu validierenden Sequenz neben D- und E-Knoten zwar P- und C-Knoten, aber keine T-Knoten erlaubt. Warum `XMLVALIDATE` in der Wurzelebene vorkommende T-Knoten (analog zu P- und C-Knoten) nicht einfach ignoriert, ist nicht einsichtig. Auch A- und N-Knoten sowie atomare Werte dürfen in der Wurzelebene einer zu validierenden Sequenz nicht auftreten. Über einen kleinen „Umweg“ lassen sich derartige Sequenzen aber dennoch validieren: Die Sequenz wird zunächst in ihre Sequenzeinträge (d. h. in die einzelnen Bäume und atomaren Werte) zerlegt. Diejenigen Bäume, die einen D- oder E-Knoten als Wurzel besitzen, werden nun mittels `XMLVALIDATE` jeweils einzeln validiert. Einen erfolgreichen Ausgang dieser Validierungen vorausgesetzt, werden die so erhaltenen validierten Bäume und die restlichen Sequenzeinträge abschließend wieder zur Gesamtsequenz zusammengesetzt.

Neben der Möglichkeit, eine Sequenz zu validieren, erlaubt es SQL/XML:2006 auch, eine Validitätsprüfung durchzuführen. Hierzu steht das Prädikat `IS VALID` zur Verfügung. Bei einer Validitätsprüfung wird (wie bei einer Validierung) getestet, ob die Sequenz den Anforderungen eines bestimmten XML-Schemas genügt. Der entscheidende Unterschied gegenüber einer Validierung besteht jedoch darin, dass *keine* Anreicherung mit Typinformationen vorgenommen wird. Während das Resultat einer (erfolgreichen) Validierung eine mit Typinformationen angereicherte Sequenz ist, liefert eine Validitätsprüfung lediglich einen (vom Ausgang der Prüfung abhängigen) Wahrheitswert zurück. Wir werden im weiteren Verlauf dieser Arbeit konsequent zwischen den Begriffen *Validierung* und *Validitätsprüfung* unterscheiden.

Entsprechend dem XQuery-Datenmodell besitzt jeder Knoten eine eindeutige Identität (vgl. Abschnitt 2.10.3). Dieses Konzept der Knotenidentität ist auch für SQL/XML:2006 relevant — z. B. bei der Übergabe einer (Knoten enthaltenden) Sequenz an eine SQL-

Funktion. Hierbei stellt sich nämlich die Frage, ob die Knoten *selbst* oder *in Kopie* übergeben werden. SQL/XML:2006 führt hierfür (in Anlehnung an vergleichbare Möglichkeiten in Programmiersprachen) zwei verschiedene Übergabemechanismen ein: *BY REF* und *BY VALUE* [Zem04a]. Bei Nutzung von *BY REF* erfolgt eine Übergabe der Knoten selbst — es werden also keine Kopien der Knoten erstellt. Wird hingegen der Übergabemechanismus *BY VALUE* verwendet, so werden nicht die Knoten selbst, sondern Kopien der Knoten (mit neuen Knotenidentitäten) übergeben.

2.12 XML-Unterstützung heutiger relationaler Datenbankprodukte

Der Wunsch, XML-Daten und „herkömmliche“ SQL-Daten *gemeinsam* verwalten und *integriert* auswerten zu können, wurde (außer von den SQL-Normungsgremien) auch von den Herstellern relationaler Datenbankmanagementsysteme *(RDBMS)* erkannt: Oracle, Microsoft und IBM — die drei Marktführer auf diesem Gebiet — haben ihre RDBMS-Produkte um XML-Funktionalität erweitert.

Wir werden im Folgenden aufzeigen, inwieweit XML-Werte in den aktuellen Versionen der einzelnen Produkte unterstützt werden. Hierzu betrachten wir *Oracle 11g Release 1* (Abschnitt 2.12.1), Microsoft *SQL Server 2005* (Abschnitt 2.12.2) sowie IBM *DB2 Universal Database 9.5* (Abschnitt 2.12.3). Im Anschluss gibt Abschnitt 2.12.4 einen vergleichenden Überblick über diese drei Produkte. Im Abschnitt 2.12.5 folgt abschließend ein kurzes Fazit.

2.12.1 Oracle 11g Release 1

Zur Verwaltung von XML-Werten stellt Oracle den Built-in-Datentyp `XMLType` bereit [ORA07c, ORA07b]. In Spalten dieses Datentyps sind (außer SQL-Nullwerten) ausschließlich wohlgeformte XML-Dokumente speicherbar. Sequenzen, die aus *mehreren* Sequenzeinträgen bestehen, lassen sich mit Oracle somit nicht handhaben — eine wesentliche Einschränkung gegenüber SQL/XML:2006.

Die in der SQL-Norm vorgesehene Verfeinerung in die XML-Untertypen `XML(SEQUENCE)`, `XML(CONTENT)` und `XML(DOCUMENT)` wird von Oracle nicht angeboten. Dies ist eine Konsequenz daraus, dass sich Oracle auf wohlgeformte XML-Dokumente beschränkt.

Oracle bietet die Möglichkeit, ein XML-Schema an eine `XMLType`-Spalte zu binden. Beim Einfügen von Werten in eine solche Spalte führt Oracle dann eine Validitätsprüfung durch [ORA07c], sodass die Gültigkeit der eingefügten XML-Dokumente garantiert ist. Dieses Vorgehen kann damit als Ersatz für den SQL/XML:2006-Typ `XML(DOCUMENT(XMLSCHEMA))` angesehen werden.

Eine (vom möglicherweise an die `XMLType`-Spalte gebundenen XML-Schema unabhängige) Validitätsprüfung ist mit Hilfe der PL/SQL-Funktion *XMLIsValid* möglich. Das von der SQL-Norm hierfür eigentlich vorgesehene Prädikat `IS VALID` steht in Oracle nicht zur Verfügung.

Mittels der PL/SQL-Prozedur *schemaValidate* lassen sich XML-Werte gegen ein XML-Schema validieren. Die von SQL/XML:2006 zu diesem Zweck eingeführte `XMLVALIDATE`-Funktion wird von Oracle nicht angeboten [ORA07a].

Um XQuery-Anfragen innerhalb von SQL formulieren und auswerten zu können, stellt Oracle (wie von SQL/XML:2006 gefordert) die Funktion `XMLQUERY` bereit. Im Gegensatz zur SQL-Norm, welche für die Übergabe von XML-Werten an Funktionen die beiden Übergabemechanismen *BY REF* und *BY VALUE* anbietet, besteht in Oracle keine solche Wahlmöglichkeit — es wird stets die Variante *BY VALUE* genutzt.

2.12.2 MS SQL Server 2005

Für die Speicherung von XML-Werten bietet der SQL Server den Built-in-Datentyp `XML` an [Mic07b, Pet06, Rys05]. Als Instanzen dieses Datentyps sind (neben SQL-Nullwerten) XML-Dokumente im Sinne des XQuery-Datenmodells erlaubt. Außer wohlgeformten XML-Dokumenten können also auch XML-Dokumente gehandhabt werden, die nicht *genau ein* Wurzelelement besitzen. Die Verwaltung von Sequenzen, die *mehrere* Sequenzeinträge enthalten, ist jedoch nicht möglich.

Die von SQL/XML:2006 vorgegebene Verfeinerung in XML-Unter(unter)typen wird vom SQL Server zumindest ansatzweise umgesetzt: Bei der Definition einer `XML`-Spalte kann ein XML-Schema spezifiziert werden, dem die in die Spalte einzufügenden XML-Dokumente genügen müssen. Dabei kann ferner entschieden werden, ob XML-Dokumente im Sinne des XQuery-Datenmodells oder ausschließlich wohlgeformte XML-Dokumente erlaubt sein sollen. Es handelt sich hierbei also um Entsprechungen der SQL/XML:2006-Unteruntertypen `XML(CONTENT(XMLSCHEMA))` und `XML(DOCUMENT(XMLSCHEMA))`. Anzumerken ist allerdings, dass der SQL Server zahlreiche Einschränkungen bezüglich der verwendbaren XML-Schemata vorsieht [Mic07a].

Die Festlegung, dass es sich um *wohlgeformte* XML-Dokumente handeln soll, ist beim SQL Server jedoch nur in Verbindung mit der Angabe eines XML-Schemas möglich — SQL/XML:2006-Typen wie `XML(DOCUMENT(UNTYPED))` oder `XML(DOCUMENT(ANY))` werden somit nicht bereitgestellt. Da der SQL Server keine Sequenzen mit mehreren Sequenzeinträgen zulässt, wird auch der Typ `XML(SEQUENCE)` nicht unterstützt.

Der SQL Server stellt weder das von der SQL-Norm für eine Validitätsprüfung vorgesehene Prädikat `IS VALID` noch die für eine Validierung geforderte Funktion `XMLVALIDATE` zur Verfügung. Eine (vom möglicherweise für die `XML`-Spalte festgelegten XML-Schema unabhängige) Validierung ist — im Gegensatz zu einer reinen Validitätsprüfung — jedoch über den Umweg einer Typkonvertierung mittels `CAST`-Funktion möglich. Als Zieldatentyp dient dabei ein `XML`-Typ, an den das gewünschte XML-Schema gebunden wird [Rys05]. Wie bereits erwähnt, existieren aber etliche Restriktionen bezüglich der nutzbaren XML-Schemata.

Der SQL Server erlaubt es, XQuery-Anfragen innerhalb von SQL zu formulieren und auszuwerten. Es wird allerdings nur eine *eingeschränkte* Fassung von XQuery unterstützt, die sich an einem XQuery-Entwurf aus dem Jahr 2004 orientiert [Mic07c]. Zur Einbettung von XQuery-Anfragen ist die `XML`-Typ-Methode *query* zu verwenden — die von SQL/XML:2006 hierfür eigentlich vorgesehene Funktion `XMLQUERY` wird nicht bereitgestellt. Die von der SQL-Norm geforderte Wahlmöglichkeit zwischen den beiden Übergabemechanismen *BY REF* und *BY VALUE* wird vom SQL Server nicht angeboten.

2.12.3 IBM DB2 Universal Database 9.5

Zur Handhabung von XML-Werten stellt DB2 den Built-in-Datentyp `XML` zur Verfügung [IBM07a, ÖCKM06, IBM05]. In Spalten dieses Datentyps können (außer SQL-Nullwerten) ausschließlich wohlgeformte XML-Dokumente abgelegt werden. Sequenzen mit *mehreren* Sequenzeinträgen sind somit nicht speicherbar.

Die von SQL/XML:2006 vorgesehene Verfeinerung in verschiedene XML-Unter(unter)-typen wird von DB2 nicht unterstützt. Untertypen wie `XML(CONTENT)` und `XML(SEQUENCE)` wären (da sich DB2 auf wohlgeformte XML-Dokumente beschränkt) zwar ohnehin nicht sinnvoll, jedoch würde sich die Einführung von Typen à la `XML(DOCUMENT(UNTYPED))`, `XML(DOCUMENT(XMLSCHEMA))` und `XML(DOCUMENT(ANY))` durchaus anbieten.

Für die Validierung von XML-Werten stellt DB2 (wie von der SQL-Norm gefordert) die Funktion `XMLVALIDATE` bereit. DB2 ermöglicht damit eine „echte“ Validierung — es wird also nicht bloß überprüft, ob das XML-Dokument dem XML-Schema genügt, sondern es findet (einen positiven Ausgang der Überprüfung vorausgesetzt) auch eine Anreicherung mit Typinformationen statt [ÖCKM06]. Da in DB2 die Festlegung des bei der Validierung zu verwendenden XML-Schemas XML-Dokument-weise (und nicht spaltenweise) erfolgt, kann ein und dieselbe `XML`-Spalte neben nicht-validierten XML-Dokumenten zugleich auch XML-Dokumente enthalten, die bezüglich *verschiedener* XML-Schemata validiert wurden.

Eine reine Validitätsprüfung (die in SQL/XML:2006 mit Hilfe des Prädikats `IS VALID` möglich ist) wird in DB2 nicht angeboten. In DB2 steht zwar das Prädikat `IS VALIDATED` zur Verfügung — mit diesem lässt sich aber nur testen, ob ein XML-Dokument bereits explizit mittels `XMLVALIDATE` validiert wurde. Zur Überprüfung der Gültigkeit eines nicht explizit validierten XML-Dokuments ist dieses Prädikat nicht nutzbar. Bei dem von DB2 angebotenen `IS VALIDATED` handelt es sich somit nicht um einen adäquaten Ersatz für das SQL/XML:2006-Prädikat `IS VALID`.

DB2 bietet die Möglichkeit, XQuery-Anfragen in SQL-Anfragen einzubetten. Hierfür steht (wie von der SQL-Norm vorgesehen) die Funktion `XMLQUERY` zur Verfügung. Im Gegensatz zur SQL-Norm, wo bei der Übergabe von XML-Werten an Funktionen zwischen den beiden Übergabemechanismen *BY REF* und *BY VALUE* gewählt werden kann, erlaubt DB2 keine derartige Auswahl — es wird stets die Variante *BY REF* verwendet.

2.12.4 Vergleichender Überblick

Die in den *Abbildungen 2.28* und *2.29* dargestellten Tabellen fassen nochmals kompakt zusammen, inwieweit SQL-Norm und Produkte XML-Werte unterstützen. Für nähere Details verweisen wir auf die Abschnitte 2.11.3, 2.12.1, 2.12.2 bzw. 2.12.3. Da jedes der betrachteten Produkte bezüglich seiner XML-Fähigkeiten gewisse Stärken und Schwächen aufweist, ist es kaum möglich, eine (objektive) Produktrangfolge *(Ranking)* zu ermitteln — wir werden deshalb auch darauf verzichten.

	Datentyp	XML-Wert	*mehrere* Sequenzeinträge
SQL/XML:2006	`XML`	beliebige Sequenz[1]	✓
Oracle 11g R1	`XMLType`	wohlgeformtes XML-Dokument[1]	–
SQL Server 2005	`XML`	XML-Dokument im Sinne des XDM[1,2]	–
DB2 UDB 9.5	`XML`	wohlgeformtes XML-Dokument[1]	–

[1]oder SQL-Nullwert, [2]XDM = XQuery-Datenmodell, ✓ = zulässig, – = unzulässig

Abbildung 2.28: XML-Unterstützung in SQL-Norm und Produkten — Teil 1/2

	`XML`-Untertypen	Validierung[1] möglich?	Validitätstest[1] möglich?	XQuery-anfragen einbettbar?	Wahl zwischen *BY REF* und *BY VALUE*?
SQL/XML:2006	✓	✓	✓	✓	✓
Oracle 11g R1	–	✓[2]	✓[2]	✓	–
SQL Server 2005	(✓)[3]	(✓)[4]	–	(✓)[5]	–
DB2 UDB 9.5	–	✓	–	✓	–

✓ = ja, – = nein, [1]unabhängig vom möglicherweise an die Spalte gebundenen XML-Schema

[2]mittels proprietärer PL/SQL-Methode, [3]nur ansatzweise

[4]mittels `CAST` realisierbar, aber Einschränkungen bezüglich verwendbarer XML-Schemata

[5]mittels proprietärer `XML`-Typ-Methode, jedoch nur eingeschränkte XQuery-Fassung nutzbar

Abbildung 2.29: XML-Unterstützung in SQL-Norm und Produkten — Teil 2/2

2.12.5 Fazit

Die marktbedeutendsten relationalen DBMS-Produkte (Oracle, SQL Server, DB2) stellen heutzutage XML-Funktionalität bereit. Dies unterstreicht die Relevanz der SQL-basierten Verarbeitung von XML-Werten. Besonders eindrucksvoll ist hierbei die rasante Entwicklung auf diesem Gebiet — während beispielsweise die Version 8.2 von DB2 [IBM04a, IBM04b, Bro04] noch nicht einmal einen Built-in-Datentyp für XML-Werte anbot, gibt es seit der (teils auch als *DB2 Viper* bezeichneten) Nachfolgeversion 9.1 [Kol06, PN06, Böh06] außer einem entsprechenden Datentyp auch sehr leistungsfähige XML-relevante SQL-Funktionen.

Wir sind überzeugt, dass die DBMS-Hersteller ihre XML-Unterstützung auch in Zukunft weiter ausbauen werden, um sich so dem SQL/XML:2006-Normstand (und seinen noch zu verabschiedenden Folgeversionen) anzunähern. Für diese Einschätzung spricht u. a. das Engagement, mit dem beispielsweise IBM und Oracle die Weiterentwicklung des (für XML relevanten) SQL-Norm-Teils SQL/XML seit Jahren vorantreiben.

Obwohl die heutigen RDBMS-Produkte bereits nennenswerte XML-Fähigkeiten besitzen, bieten sie noch keine angemessene Unterstützung des SQL/XML:2006-Datentyps `XML(SEQUENCE)`. Während SQL/XML:2006 als XML-Werte ausdrücklich Sequenzen mit *mehreren* Sequenzeinträgen erlaubt (also geordnete Folgen, bestehend aus *mehreren* Bäumen und/oder atomaren Werten), beschränken sich sämtliche Produkte auf XML-Dokumente (also einzelne Bäume).

2.13 Beispielszenario Kundenkartenverwaltung

Der wichtigste Schritt bei der Weiterentwicklung von SQL/XML:2003 zu SQL/XML:2006 war der Wechsel des dem SQL-Basisdatentyp `XML` zugrunde gelegten Datenmodells vom XML-Infoset-basierten Datenmodell zum XQuery-Datenmodell. Oder anders ausgedrückt: Die wesentliche Neuerung von SQL/XML:2006 gegenüber SQL/XML:2003 besteht darin, dass als XML-Werte anstatt einzelner XML-Dokumente (d. h. einzelner Bäume) beliebige XQuery-Sequenzen (d. h. geordnete Folgen von Bäumen und/oder atomaren Werten) zulässig sind. Aber gerade diese entscheidende Neuerung wird (momentan) noch von keinem RDBMS-Produkt reflektiert. Infolgedessen existiert in der Praxis derzeit auch noch kein „echtes“, real erprobtes SQL/XML:2006-Anwendungsszenario. Als Grundlage für weitere Erläuterungen werden wir deshalb kurz ein eigenes realitätsnahes, auf SQL/XML:2006 abgestimmtes Beispielszenario einführen.

Im Mittelpunkt unseres Anwendungszenarios [Mar05] steht ein (fiktives) Unternehmen, welches *Kundenkarten* ausgibt und verwaltet. Die Kundenkarten sollen beispielsweise bei Einkäufen, Autoanmietungen oder Hotelübernachtungen einsetzbar sein, um Rabatte bzw. Sonderkonditionen zu erhalten. Die von uns gewählte Thematik besitzt eine hohe Praxisrelevanz [HW04]. Allein die *Loyalty Partner GmbH*, Herausgeberin der „PAYBACK-Karte“, verwaltet nach eigenen Angaben deutschlandweit ca. 30 Millionen Kundenkarten [Loy06].

Als Kaufanreiz können den Kunden in unserem Beispielszenario so genannte *Bonusangebote* unterbreitet werden. Ein geeignetes Bonusangebot zur Steigerung des Umsatzes von Babynahrung wäre z. B.: *„Wer für mindestens 3 Euro Spinatbrei kauft — und seine Kundenkarte vorlegt — bekommt bei einem Folgeeinkauf zwei Päckchen Waschmittel zum Preis von einem.“*

Die (i. d. R. zeitlich befristeten) Bonusangebote sollen auch einen „einlösbaren Gegenwert“ besitzen, z. B. 57 Cent bei einem späteren Einkauf im selben Supermarkt oder 32 Cent bei einem Folgeeinkauf in einem anderen (an der Aktion teilnehmenden) Geschäft. Ist der Kunde im konkreten Fall z. B. nicht am verbilligten Erwerb von Waschmittel interessiert, kann er sich stattdessen den einlösbaren Gegenwert auf einen beliebigen anderen Einkauf anrechnen lassen. Die Entscheidung, welche Alternative er nutzen will (verbilligtes Waschmittel oder einlösbaren Gegenwert) muss er dabei nicht sofort treffen — er hat damit Zeit, bis die Befristung des Bonusangebots endet. Im Gegensatz zu „traditionellen“ Kundenkartenmodellen besitzt ein Kunde in unserem Beispiel also nicht einfach einen simplen Punktestand, sondern er kann für jedes Bonusangebot separat entscheiden, ob er es direkt in Anspruch nimmt oder ob er sich stattdessen den einlösbaren Gegenwert anrechnen lässt.

Für die Verwaltung der Bonusdaten nehmen wir eine SQL/XML:2006-Tabelle an. Da die Bonusangebote sehr flexibel strukturiert sein können, bietet sich für sie eine XML-basierte Speicherung an. Um den Neuerungen von SQL/XML:2006 gerecht zu werden, unterstellen wir, dass die Bonusdaten als „echte“ Sequenzen abgelegt sind. Unter „echten“ Sequenzen wollen wir dabei XQuery-Sequenzen verstehen, die i. d. R. aus *mehreren* Sequenzeinträgen bestehen. Wir gehen im Folgenden davon aus, dass jedes von einem Kunden erworbene Bonusangebot jeweils als ein Eintrag einer solchen Sequenz repräsentiert wird.

Für eine leichtere Auswertbarkeit und bessere Handhabbarkeit der Bonusdaten setzen wir ferner voraus, dass die Bonusdaten für jeden Kunden jeweils getrennt nach der Kategorie (z. B. Dienstleistung, Einkauf, Anmietung) abgelegt sind. Unsere in *Abbildung 2.30* dargestellte Bonusdatentabelle *Kukabo (Kunden-Kategorie-Bonus-Tabelle)* besitzt damit

einen aus der Kundennummer *(KdNr)* und der Kategorie bestehenden Primärschlüssel. Die Einträge der Spalte *letzmals_aktiv* geben an, wann der Kunde in der entsprechenden Kategorie das letzte Mal aktiv war. Bei einer solchen Aktivität kann es sich z. B. um die Generierung eines Umsatzes oder um die Beteiligung an einer Kundenbefragung handeln.

Die Spalte *Bonus*, welche die Bonusangebote enthält, ist vom SQL/XML:2006-Datentyp `XML(SEQUENCE)`. In der abgebildeten *Kukabo*-Tabelle verfügt beispielsweise der Kunde mit der Kundennummer *4711* in der Kategorie *Dienstleistung* über sieben Bonusangebote — jedes Bonusangebot entspricht dabei einem Sequenzeintrag, d. h. einem Baum innerhalb der Folge von Bäumen. Wir wollen hierbei unterstellen, dass die Ordnung der Sequenzeinträge die zeitliche Reihenfolge widerspiegelt, in der die Bonusangebote erworben wurden. Der erste (d. h. linkeste) Sequenzeintrag repräsentiert somit das älteste Bonusangebot, während das zweitälteste Bonusangebot durch den zweiten Eintrag der Sequenz dargestellt wird usw.

Kukabo

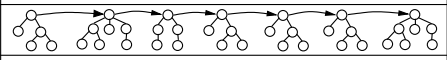
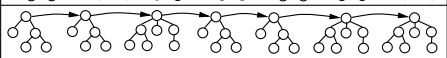
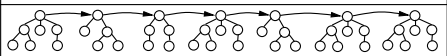
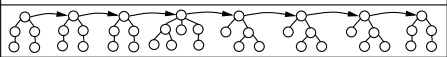
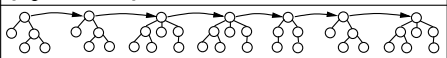
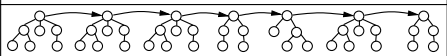
KdNr	Kategorie	Bonus	letztmals_aktiv
4711	Dienstleistung		01.06.2008
4711	Einkauf		05.07.2008
4711	Anmietung		10.02.2008
4711	Übernachtung		03.02.2008
4711	Flugbuchung		21.05.2008
4712	Dienstleistung		20.06.2008
⋮	⋮	⋮	⋮

Abbildung 2.30: Kunden-Kategorie-Bonus-Tabelle *Kukabo*

Um die Kunden in unserem Beispielszenario z. B. zur Teilnahme an einer Zufriedenheitsumfrage zu motivieren, wird ihnen die Aufwertung bestimmter Bonusangebote in Aussicht gestellt: *Wer an der Kundenbefragung teilnimmt, erhält als Belohnung einen „Dankeschön-Code“, mit dem er am Kundenterminal jeweils die letzten fünf Bonusangebote derjenigen Kategorien aufwerten kann, in denen er innerhalb der letzten drei Monate aktiv war. Bei jedem aufwertbaren Bonusangebot kann dabei jeweils zwischen der Erhöhung des einlösbaren Gegenwerts um 50 Cent und einer Verlängerung der Gültigkeit um einen Monat gewählt werden.*

Durch die dem Kunden angebotene Wahlmöglichkeit kann die Aufwertung der betroffenen Bonusangebote nicht automatisch erfolgen. Stattdessen muss der Kunde für jedes aufwertbare Bonusangebot jeweils separat entscheiden, welche Art der Aufwertung er in Anspruch nehmen will. Hierzu müssen die entsprechenden Bonusangebote in das zur Aufwertung genutzte Anwendungsprogramm übertragen werden. Wir werden im Kapitel 4 ein Verfahren vorstellen, mit dessen Hilfe eine derartige Übertragung möglich ist.

Kapitel 3

Typed-value-orientierte Repräsentation

Wir werden in diesem Kapitel die *typed-value-orientierte Repräsentation* [MR05, Yan06] von Sequenzen einführen, welche als Grundlage für die im Kapitel 4 beschriebene Sequenzcursor-basierte Verarbeitung dient. Nach einer Motivation der typed-value-orientierten Repräsentation im Abschnitt 3.1 gehen wir im Abschnitt 3.2 auf Konventionen der graphischen Darstellung ein. Anschließend erläutern wir in den Abschnitten 3.3 und 3.4, wie die getypten Werte von Element- bzw. Attributknoten repräsentiert werden. Der Repräsentation der getypten Werte anderer Knotenarten widmet sich dann Abschnitt 3.5. Im Anschluss beschreibt Abschnitt 3.6 den Umgang mit atomaren Werten, die als Sequenzeinträge auftreten. Die Handhabung von XML-Namensräumen erfordert besondere Aufmerksamkeit. Wir gehen hierauf im Abschnitt 3.7 detaillierter ein. In den Abschnitten 3.8 und 3.9 folgen ein Fazit sowie ein abschließendes Beispiel.

3.1 Motivation

Das Grundprinzip der Sequenzcursor-basierten Verarbeitung (Kapitel 4) besteht darin, mit Hilfe von Sequenzcursorn Ausschnitte der im aktuellen Ergebnistupel enthaltenen (XQuery-)Sequenzen zu definieren, welche dann ins Anwendungsprogramm übertragen und dort lokal verarbeitet werden. Im Abschnitt 3.1.1 zeigen wir, dass sich die „traditionelle“ Sequenzrepräsentation *nicht* eignet, um als Basis dieses Verfahrens genutzt zu werden. Unter der traditionellen Sequenzrepräsentation verstehen wir dabei die Repräsentation einer Sequenz als geordnete Folge von Bäumen und/oder atomaren Werten (vgl. Abschnitt 2.10.3), wobei die Bäume aus den im Abschnitt 2.10.2 beschriebenen Knoten bestehen.

Da sich die traditionelle Sequenzrepräsentation nicht als Grundlage der Sequenzcursor-basierten Verarbeitung eignet, sollte stattdessen eine abgewandelte (und auf die Sequenzcursor-basierte Verarbeitung abgestimmte) Variante genutzt werden. Den Anforderungen an eine derartige Sequenzrepräsentation widmet sich Abschnitt 3.1.2.

Im weiteren Verlauf des aktuellen Kapitels führen wir dann die typed-value-orientierte Repräsentation von Sequenzen ein, welche allen Anforderungen genügt, um als Basis der Sequenzcursor-basierten Verarbeitung verwendet zu werden. Es handelt sich dabei um

eine von uns entwickelte Repräsentationsvariante, die an die speziellen Erfordernisse der Sequenzcursor-basierten Verarbeitung angepasst ist.

3.1.1 Nachteile der traditionellen Sequenzrepräsentation

Eine Nutzung der traditionellen Sequenzrepräsentation als Grundlage für die Sequenzcursor-basierte Verarbeitung wäre mit diversen Nachteilen verbunden. Wir werden im Folgenden jeden dieser Nachteile jeweils anhand eines konkreten Beispiels diskutieren.

In den zu diesen Beispielen gehörenden Abbildungen (wie auch in den Abbildungen der Abschnitte 3.2 bis 3.6) haben wir auf die Darstellung von N-Knoten verzichtet, da diese Knoten für die entsprechenden Erläuterungen nicht relevant sind. Eine ausführliche Betrachtung der N-Knoten erfolgt später im Abschnitt 3.7.

Jeder in den folgenden Beispielen vorkommende E-Knoten besitzt E- oder T-Kindknoten, sodass seine *nilled*-Eigenschaft zwangsläufig den Wert *false* hat (vgl. Abschnitt 2.10.2.4). Wir werden dies bei unseren Beispielen als bekannt voraussetzen und nicht jedes Mal explizit erwähnen.

- *Im Vaterknoten „versteckte“ Typinformationen*

  In Anlehnung an unser XML-Dokument aus Abschnitt 2.7.1 und unser im Abschnitt 2.7.3 angedeutetes XML-Schema betrachten wir exemplarisch einen E-Knoten namens *Preis*, welcher vom Typ *PreisTyp* ist und einen T-Kindknoten mit dem Inhalt *“21.05“* besitzt *(Abbildung 3.1)*. Auf die Darstellung des zugeordneten A-Knotens haben wir verzichtet, da dieser Knoten für unsere Betrachtungen nicht von Bedeutung ist.

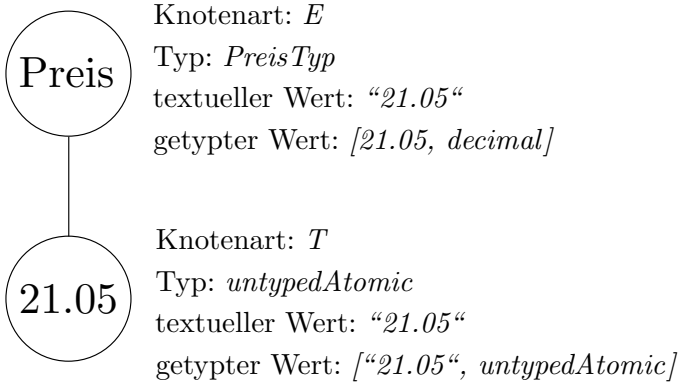


Abbildung 3.1: E-Knoten *Preis* mit Kindknoten

  Der textuelle Wert des E-Knotens lautet *“21.05“*, sein getypter Wert *[21.05, decimal]* (vgl. Abschnitt 2.10.2.5). Der T-Knoten ist (wie alle T-Knoten) vom Typ *untypedAtomic* (vgl. Abschnitt 2.10.2.4). Bei seinem textuellen Wert handelt es sich um *“21.05“*, bei seinem getypten Wert um *[“21.05“, untypedAtomic]* (vgl. Abschnitt 2.10.2.5). Der T-Knoten selbst besitzt also *keinerlei* Typinformation, die besagt, dass es sich bei seinem Inhalt um eine Dezimalzahl handelt. Die entsprechende Typinformation ist stattdessen im getypten Wert seines Vaterknotens „versteckt“.

Wir betrachten zunächst den Fall, dass sowohl der T-Knoten als auch sein Vaterknoten Teil eines ins Anwendungsprogramm übertragenen Sequenzausschnitts sind. Soll der Inhalt des T-Knotens lokal verarbeitet werden, so muss — zur Ermittlung der Typinformation — auch auf den Vaterknoten (also den E-Knoten namens *Preis*) zugegriffen werden. Zur Verarbeitung eines einzelnen Werts ist also ein Zugriff auf zwei verschiedene Knoten erforderlich. Hierdurch wird das lokale Arbeiten auf dem Sequenzausschnitt erheblich erschwert.

Wir widmen uns nun dem (kritischeren) Fall, dass zwar der T-Knoten, nicht aber sein Vaterknoten Teil des Sequenzausschnitts ist. Hierbei besteht das Problem darin, dass für den T-Knoteninhalt lokal keinerlei Typinformationen verfügbar sind. Damit ist eine angemessene lokale Verarbeitung des T-Knoteninhalts i. d. R. nicht möglich.

- *Anomalien in Folge von Redundanz*

  Wir betrachten einen T-Knoten mit dem Inhalt *"Jena"*, der einen E-Vaterknoten namens *Stadt* besitzt, welcher wiederum einen E-Vaterknoten namens *Hauptwohnsitz* hat, welcher wiederum einen E-Vaterknoten namens *Wohnsitz* besitzt *(Abbildung 3.2)*. Die drei E-Knoten seien dabei vom Typ *anyType*. Der T-Knoten ist (wie sämtliche T-Knoten) vom Typ *untypedAtomic*.

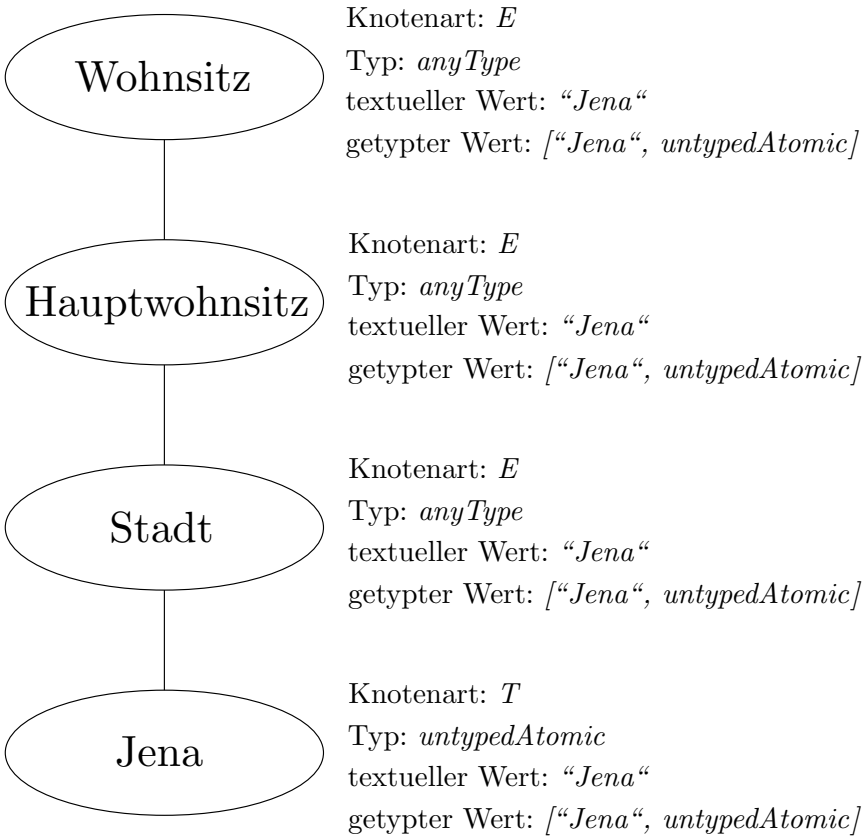


Abbildung 3.2: E-Knoten *Wohnsitz* mit Nachfahrenknoten

Jeder der vier Knoten besitzt damit *"Jena"* als textuellen Wert und *["Jena", untypedAtomic]* als getypten Wert (vgl. Abschnitt 2.10.2.5). Der Inhalt des T-Knotens

tritt also redundant in den textuellen und getypten Werten aller drei E-Knoten auf.

Wir unterstellen im Folgenden, dass alle vier Knoten als Teil eines Sequenzausschnitts ins Anwendungsprogramm übertragen wurden, um dort lokal verarbeitet zu werden. Soll nun der Inhalt des T-Knotens lokal geändert werden (die Zeichenkette *"Jena"* soll beispielsweise durch *"Erfurt"* ersetzt werden), so ist — um Inkonsistenzen zu vermeiden — auch eine Anpassung der textuellen und getypten Werte der drei E-Knoten erforderlich. Obwohl also lediglich der Inhalt eines einzelnen Knotens geändert werden soll, sind die Knoteneigenschaften von insgesamt vier Knoten zu ändern. Diese Anomalie (welche den vom relationalen Modell bekannten *Änderungsanomalien* ähnelt) erschwert die lokale Verarbeitung des Sequenzausschnitts.

- *Aufsplittung atomarer Werte*

  Für die folgenden Betrachtungen setzen wir einen E-Knoten namens *bezahlt* voraus, der vom Typ *boolean* ist und drei Kindknoten besitzt. Bei seinem linkesten Kind handle es sich um einen T-Knoten mit Inhalt *"tr"*, bei seinem mittleren Kind um einen C-Knoten mit Inhalt *"bar"* und bei seinem rechtesten Kind um einen T-Knoten mit Inhalt *"ue" (Abbildung 3.3).*

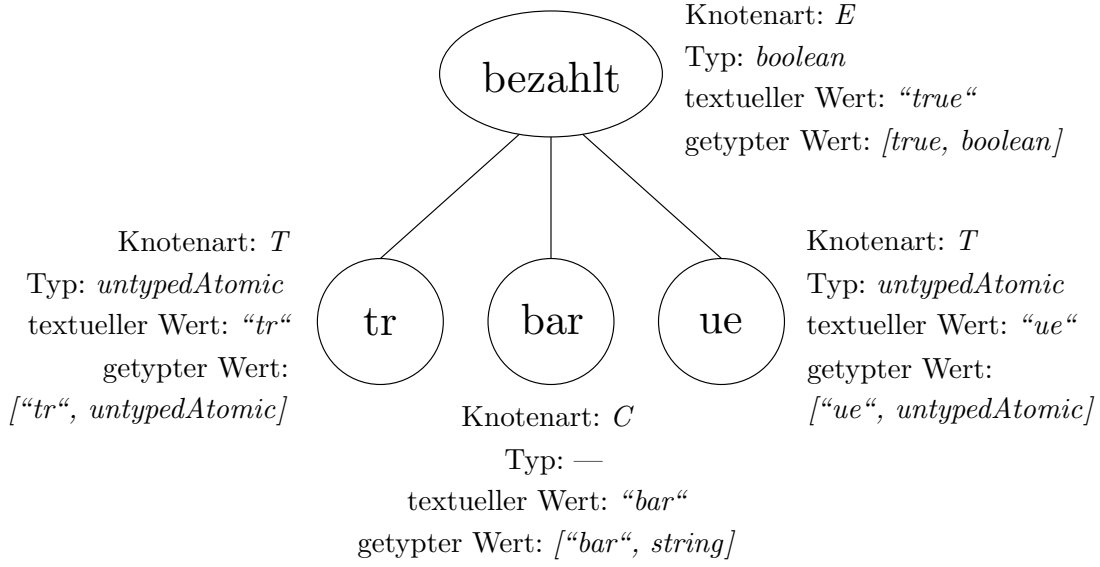


Abbildung 3.3: E-Knoten *bezahlt* mit Kindknoten

Der textuelle Wert des E-Knotens lautet somit *"true"*, sein getypter Wert *[true, boolean]* (vgl. Abschnitt 2.10.2.5). Der C-Knoten ist (aufgrund seiner Knotenart) typlos (vgl. Abschnitt 2.10.2.4). Er besitzt *"bar"* als textuellen Wert und *["bar", string]* als getypten Wert. Die beiden T-Knoten haben (wie alle T-Knoten) den Typ *untypedAtomic*. Bei ihren textuellen Werten handelt es sich um *"tr"* bzw. *"ue"*, bei ihren getypten Werten um *["tr", untypedAtomic]* bzw. *["ue", untypedAtomic]*. Jeder der beiden T-Knoten repräsentiert also jeweils nur einen *Teil* des atomaren Werts *[true, boolean]*.

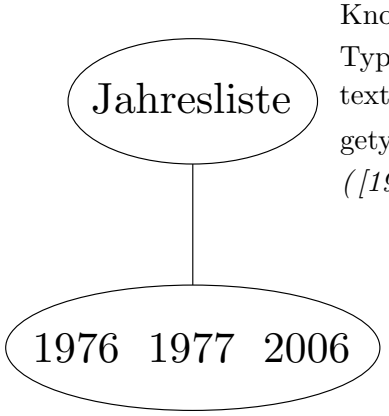
Wir nehmen zunächst an, dass zwar der T-Knoten *"tr"*, nicht jedoch der T-Knoten *"ue"* zum lokal zu verarbeitenden Sequenzausschnitt gehört. Somit ist im Anwendungsprogramm nur ein *Teil* des atomaren (und damit eigentlich „unteilbaren")

Werts *[true, boolean]* verfügbar. Aus semantischer Sicht ist es i. d. R. allerdings nicht sinnvoll, Teile eines atomaren Werts separat zu verarbeiten. Somit ist eine adäquate lokale Verarbeitung des T-Knoteninhalts *"tr"* nicht möglich.

Wir betrachten jetzt den Fall, dass *beide* T-Knoten zum Sequenzausschnitt gehören. Soll nun der Wert *true* in *false* umgeändert werden, so sind davon beide T-Knoten betroffen. Zur Änderung eines einzelnen atomaren Werts ist also ein ändernder Zugriff auf zwei verschiedene T-Knoten nötig. Hierdurch wird das lokale Arbeiten auf dem Sequenzausschnitt deutlich erschwert.

- *Erschwerter Zugriff auf einzelne atomare Werte*

  Wir setzen im Folgenden einen E-Knoten namens *Jahresliste* voraus, der vom Typ *JahreslistenTyp* ist und einen T-Kindknoten mit Inhalt *"1976 1977 2006"* hat *(Abbildung 3.4)*. Bei *JahreslistenTyp* handle es sich um einen Listentyp, der basierend auf dem im Abschnitt 2.10.2.5 eingeführten atomaren Typ *JahresTyp* definiert sei.



Knotenart: *E*
Typ: *JahreslistenTyp*
textueller Wert: *"1976 1977 2006"*
getypter Wert:
([1976, JahresTyp], [1977, JahresTyp], [2006, JahresTyp])

Knotenart: *T*
Typ: *untypedAtomic*
textueller Wert: *"1976 1977 2006"*
getypter Wert: *["1976 1977 2006", untypedAtomic]*

Abbildung 3.4: E-Knoten *Jahresliste* mit Kindknoten

Der T-Knoten (der wie sämtliche T-Knoten vom Typ *untypedAtomic* ist) besitzt *"1976 1977 2006"* als textuellen Wert und *["1976 1977 2006", untypedAtomic]* als getypten Wert. Der textuelle Wert des E-Knotens lautet *"1976 1977 2006"*, sein getypter Wert *([1976, JahresTyp], [1977, JahresTyp], [2006, JahresTyp])* (vgl. Abschnitt 2.10.2.5). Der getypte Wert des E-Knotens ist also eine Folge, die aus drei atomaren Werten des Typs *JahresTyp* besteht. Es sei hier ausdrücklich hervorgehoben, dass diese drei atomaren Werte *gemeinsam* durch einen *einzelnen* T-Knoten (welcher zugleich der einzige T-Knoten unseres Beispiels ist) repräsentiert werden.

Wir unterstellen jetzt, dass der T-Knoten Teil eines ins Anwendungsprogramm übertragenen Sequenzausschnitts ist. Soll nun im Rahmen der lokalen Verarbeitung beispielsweise die mittlere Jahreszahl von *1977* auf *1995* erhöht werden, so ist dies nur „umständlich" unter Zuhilfenahme von Operationen zur Zeichenkettenmanipulation möglich — das entsprechende Teilstück der als T-Knoteninhalt dienenden Zeichenkette *"1976 1977 2006"* muss durch den neuen Wert ersetzt werden. Die Ursache für den erschwerten Zugriff auf einzelne atomare Werte liegt hier also darin, dass mehrere atomare Werte gemeinsam durch ein und denselben T-Knoten repräsentiert werden.

- *Erschwerte Auswahl einzelner atomarer Werte*

  Wir betrachten erneut den E-Knoten namens *Jahresliste* aus Abbildung 3.4, bei dessen getypten Wert es sich um eine Folge aus den drei atomaren Werten *[1976, JahresTyp]*, *[1977, JahresTyp]* und *[2006, JahresTyp]* handelt.

  Wir nehmen nun an, dass für die lokale Verarbeitung im Anwendungsprogramm nur die erste (d. h. linkeste) Jahreszahl benötigt wird. Um eine Übertragung (und lokale Verwaltung) überflüssiger Daten zu vermeiden, sollte der ins Anwendungsprogramm zu übertragende Sequenzausschnitt folglich nur den atomaren Wert *[1976, JahresTyp]*, nicht aber die atomaren Werte *[1977, JahresTyp]* und *[2006, JahresTyp]* enthalten.

  Da alle drei atomaren Werte gemeinsam durch ein und denselben T-Knoten repräsentiert werden, darf somit nur ein *Teil* dieses T-Knotens zum Sequenzausschnitt gehören. Dies bedeutet, dass der entsprechende Knotenteil bei der Definition des Sequenzausschnitts gezielt ausgewählt werden muss. Eine derartige gezielte Selektion von Knotenteilen ist jedoch deutlich komplizierter als die ausschließliche Auswahl kompletter Knoten.

  Die Auswahl eines einzelnen atomaren Werts wird in unserem Beispiel also dadurch erheblich erschwert, dass bei der Festlegung des Sequenzausschnitts gezielt ein *Teil* eines Knotens ausgewählt werden muss. Die eigentliche Ursache hierfür ist wiederum, dass der betrachtete T-Knoten zugleich *mehrere* atomare Werte repräsentiert.

Aufgrund der zuvor aufgeführten Nachteile werden wir die traditionelle Sequenzrepräsentation *nicht* als Basis der Sequenzcursor-basierten Verarbeitung verwenden.

3.1.2 Anforderungen an eine abgewandelte Sequenzrepräsentation

An eine auf die Sequenzcursor-basierte Verarbeitung abgestimmte Sequenzrepräsentation ergeben sich die folgenden Anforderungen. Die ersten fünf dieser Forderungen sind eine unmittelbare Konsequenz aus den im Abschnitt 3.1.1 durchgeführten Betrachtungen.

- *Keine „versteckten“ Typinformationen*

  Typinformationen sollten nicht im Vaterknoten (oder in anderen Knoten) versteckt werden. Eine Typinformation sollte stattdessen stets demjenigen Knoten zugeordnet sein, der auch den zugehörigen Wert enthält.

- *Vermeidung von Anomalien*

  Durch Redundanz verursachte Anomalien sollten weitestgehend vermieden werden. Um dies zu erreichen, sollte (sofern möglich) jede Information jeweils nur an einer Stelle (also nur von einem Knoten) vorgehalten werden.

- *Keine Aufsplittung atomarer Werte*

  Atomare Werte sollten (wie es bereits ihre Bezeichnung nahelegt) als *unteilbare* Werte aufgefasst werden. Insbesondere sollte ein atomarer Wert nicht auf mehrere Knoten aufgesplittet werden.

- *Einfacher Zugriff auf einzelne atomare Werte*

  Die getypten Werte von A- und E-Knoten können aus *mehreren* atomaren Werten bestehen. Es sollte unkompliziert möglich sein, auf die einzelnen atomaren Werte zuzugreifen.

- *Einfache Auswahl einzelner atomarer Werte*

  Bei der Definition eines Sequenzausschnitts sollte es unkompliziert möglich sein, einen einzelnen atomaren Wert gezielt auszuwählen. Für eine derartige gezielte Auswahl sollte es insbesondere *nicht* erforderlich sein, einen Knoten*teil* zu selektieren.

- *Automatische Transformierbarkeit*

  Die Sequenzcursor-basierte Verarbeitung sollte auch auf Sequenzen anwendbar sein, die (ursprünglich) in traditioneller Sequenzrepräsentation vorliegen. Traditionell repräsentierte Sequenzen sollten deshalb automatisch in die auf die Sequenzcursor-basierte Verarbeitung abgestimmte Sequenzrepräsentation überführt werden können. Eine Rücktransformation sollte ebenfalls möglich sein.

- *Äquivalenz des Informationsgehalts*

  Um die zuvor geforderte automatische Transformierbarkeit zu gewährleisten, sollte die auf die Sequenzcursor-basierte Verarbeitung abgestimmte Sequenzrepräsentation dieselben Informationen enthalten wie die traditionelle Sequenzrepräsentation. Die entsprechenden Informationen können dabei jedoch auf eine andere Art und Weise dargestellt sein.

Die typed-value-orientierte Repräsentation von Sequenzen, die wir in den Abschnitten 3.3 bis 3.9 detailliert behandeln, erfüllt (im Wesentlichen) alle zuvor genannten Forderungen und wird von uns deshalb als Grundlage für die Sequenzcursor-basierte Verarbeitung genutzt.

3.2 Konventionen der graphischen Darstellung

Bevor wir die typed-value-orientierte Repräsentation von Sequenzen im weiteren Verlauf des aktuellen Kapitels einführen, wollen wir zunächst auf einige Konventionen zur graphischen Darstellung eingehen. Die Ausführungen dieses Abschnitts gelten dabei jedoch nicht nur für Sequenzen in typed-value-orientierter Repräsentation, sondern auch für traditionell repräsentierte Sequenzen.

Die W3C-Spezifikation des XQuery-Datenmodells [W3C07e] enthält leider keine verbindlichen Regeln für die graphische Darstellung von (traditionell repräsentierten) Sequenzen. Auch in der Literatur hat sich bisher noch keine einheitliche (und allgemein anerkannte) Darstellungsform etabliert. Wir werden im Fortgang dieser Arbeit deshalb eine selbst entwickelte graphische Darstellungsvariante nutzen, die wir im Folgenden kurz beschreiben.

Jeder Knoten wird bei uns durch einen Kreis dargestellt, an dem oben rechts die Knotenart (*D*, *E*, *A*, *N*, *P*, *C* bzw. *T*) vermerkt ist. Atomare Werte werden ebenfalls als Kreise visualisiert — diese sind oben rechts mit dem Buchstaben *V* (für *atomic value*) gekennzeichnet. *Abbildung 3.5* zeigt jeweils ein Beispiel für einen atomaren Wert, einen D-Knoten, einen E-Knoten sowie einen T-Knoten.

Abbildung 3.5: Beispiel für einen atomaren Wert und drei Knoten

Bei atomaren Werten (z. B. *[7, integer]*) wird der eigentliche Wert (hier: *7*) innerhalb des Kreises aufgeführt, während der Typname (hier: *integer*) unten rechts am Kreis vermerkt wird (vgl. Abbildung 3.5). Bei der Darstellung von Knoten hängt die Beschriftung *im* Kreis bzw. *unten rechts* am Kreis von der jeweiligen Knotenart ab.

Bei D-Knoten wird weder im Kreis noch unten rechts am Kreis eine Beschriftung vorgenommen. Kreise, die E- oder A-Knoten darstellen, enthalten im Kreisinnern den Knotennamen und sind unten rechts mit dem Typ des Knotens gekennzeichnet. Bei N-Knoten wird innerhalb des Kreises der Namensraumname aufgeführt — das zugehörige Namensraumpräfix wird unten rechts am Kreis vermerkt. Für P-, C- und T-Knoten entspricht die Beschriftung im Kreis dem Knoteninhalt. Während bei P-Knoten unten rechts am Kreis das Verarbeitungsanweisungsziel aufgeführt wird, besitzen Kreise, die C- oder T-Knoten darstellen, unten rechts keine Beschriftung. *Abbildung 3.6* fasst die Beschriftung der Kreise nochmals kurz zusammen.

<table>
<tr><th>Beschriftung
im Kreis</th><th>dargestelltes
Objekt</th><th>Beschriftung
oben rechts</th><th>Beschriftung
unten rechts</th></tr>
<tr><td>— (keine)</td><td>D-Knoten</td><td>D</td><td>— (keine)</td></tr>
<tr><td rowspan="2">Knotenname</td><td>E-Knoten</td><td>E</td><td rowspan="2">Typ des Knotens</td></tr>
<tr><td>A-Knoten</td><td>A</td></tr>
<tr><td>Namensraumname</td><td>N-Knoten</td><td>N</td><td>Namensraumpräfix</td></tr>
<tr><td rowspan="3">Knoteninhalt</td><td>P-Knoten</td><td>P</td><td>Verarbeitungsanweisungsziel</td></tr>
<tr><td>C-Knoten</td><td>C</td><td rowspan="2">— (keine)</td></tr>
<tr><td>T-Knoten</td><td>T</td></tr>
<tr><td>eigentlicher Wert</td><td>atomarer Wert</td><td>V</td><td>Typname</td></tr>
</table>

Abbildung 3.6: Darstellung von Knoten und atomaren Werten

Wenn die für das Innere eines Kreises vorgesehene Beschriftung relativ lang ist, kann anstelle des Kreises auch ein Oval (Ellipse) verwendet werden. Aus semantischer Sicht ist es allerdings völlig unerheblich, ob ein Knoten bzw. ein atomarer Wert als Kreis oder als Oval dargestellt wird.

Bei der Bestimmung des getypten Werts (vgl. Abschnitt 2.10.2.5) haben Leerzeichen eine besondere Bedeutung: Besitzt ein E-Knoten (oder ein A-Knoten) einen Listentyp, so werden innerhalb des textuellen Werts auftretende Leerzeichen als Trennzeichen aufgefasst. Ergibt sich der textuelle Wert des E-Knotens durch Verkettung *mehrerer* T-Knoteninhalte (weil der E-Knoten mehrere T-Kindknoten hat), so ist es folglich relevant, ob diese T-Knoteninhalte mit einem Leerzeichen beginnen bzw. enden.

Bei der graphischen Darstellung von Sequenzen werden wir jedes im Inhalt eines T-Knotens

vorkommende Leerzeichen besonders hervorheben, indem wir es durch einen Unterstrich (_) repräsentieren. Damit soll sichergestellt werden, dass insbesondere Leerzeichen, die am Anfang oder am Ende eines T-Knoteninhalts auftreten (und für die Bestimmung des getypten Werts sehr wichtig sind), nicht „übersehen“ werden.

Abbildung 3.7 zeigt die graphische Darstellung einer traditionell repräsentierten Sequenz mit vier Sequenzeinträgen. Die einzelnen Sequenzeinträge sind (unter Berücksichtigung ihrer Reihenfolge) durch Pfeile miteinander verbunden (vgl. Abschnitt 2.10.3). Bis auf den zweiten Sequenzeintrag — bei dem es sich um den atomaren Wert *[7, integer]* handelt — entsprechen die Sequenzeinträge den im Abschnitt 3.1.1 betrachteten Bäumen mit den Wurzelknoten *Preis*, *bezahlt* bzw. *Jahresliste*. Beim Baum mit dem Wurzelknoten *Preis* wurde hier (im Gegensatz zur Abbildung 3.1) allerdings nicht auf die Darstellung des zugeordneten A-Knotens *MWSt-Satz* verzichtet. Während *beidseitige* Vater-Kind-Beziehungen zwischen Knoten mit Hilfe von durchgehenden Kanten dargestellt sind, werden *einseitige* Vater-Kind-Beziehungen (vgl. Abschnitt 2.10.2.2) als gestrichelte Kanten repräsentiert.

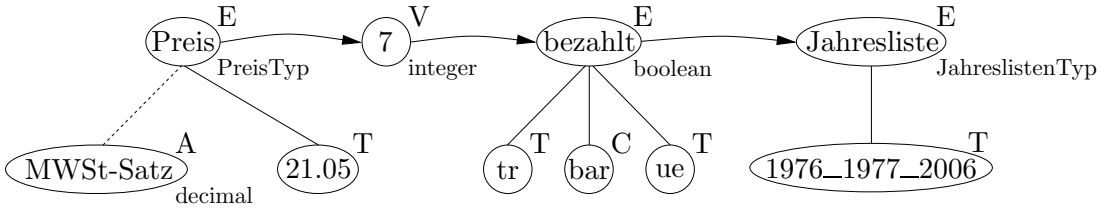


Abbildung 3.7: Beispiel für eine traditionell repräsentierte Sequenz

3.3 Repräsentation der getypten Werte von Elementknoten

Beim getypten Wert eines E-Knotens handelt es sich — sofern er definiert ist — stets um eine Folge aus keinem, einem[1] oder mehreren atomaren Werten (vgl. Abschnitt 2.10.2.5). Ausgehend hiervon unterscheiden wir im Folgenden fünf Fälle, die wir als Grundlage für eine Klassifikation der E-Knoten in *nicht-typbare*, *bedingt-typbare* sowie *echt-typbare* E-Knoten nutzen [Jat07]. Die von uns bezüglich des getypten Werts vorgenommene Fallunterscheidung wird in *Abbildung 3.8* überblicksweise veranschaulicht.

Ist der getypte Wert des E-Knotens nicht definiert *(Fall 1)* oder handelt es sich bei ihm um eine leere Folge *(Fall 2)*, so bezeichnen wir den E-Knoten als *nicht-typbar*. E-Knoten, die als getypten Wert einen einzelnen atomaren Wert vom Typ *untypedAtomic* besitzen *(Fall 3)*, werden als *bedingt-typbar* bezeichnet. Ist der getypte Wert des E-Knotens ein einzelner atomarer Wert mit einem von *untypedAtomic* abweichenden Typ *(Fall 4)* oder besteht er aus *mehreren* atomaren Werten *(Fall 5)*, so bezeichnen wir den E-Knoten als *echt-typbar*. *Abbildung 3.9* fasst die Klassifikation der E-Knoten nochmals kurz zusammen.

[1] Das XQuery-Datenmodell unterscheidet nicht zwischen einem einzelnen atomaren Wert und einer Folge, die genau diesen einen atomaren Wert enthält.

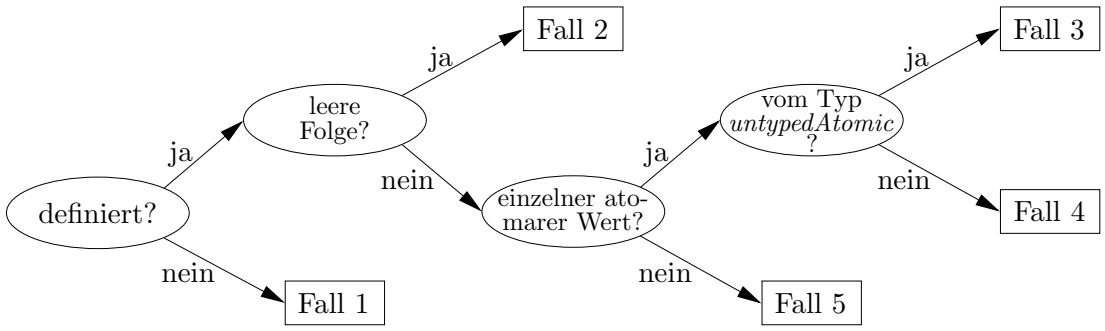


Abbildung 3.8: Fallunterscheidung bezüglich des getypten Werts von E-Knoten

Fall	getypter Wert	Klassifikation
Fall 1	nicht definiert	nicht-typbar
Fall 2	leere Folge	
Fall 3	einzelner atomarer Wert vom Typ *untypedAtomic*	bedingt-typbar
Fall 4	einzelner atomarer Wert mit einem von *untypedAtomic* abweichenden Typ	echt-typbar
Fall 5	Folge aus mehreren atomaren Werten	

Abbildung 3.9: Klassifikation der E-Knoten

Basierend auf der zuvor eingeführten Klassifikation beschreiben wir im Folgenden, wie die getypten Werte von E-Knoten im Rahmen der typed-value-orientierten Repräsentation dargestellt werden. Im Abschnitt 3.3.1 betrachten wir zunächst *echt-typbare* E-Knoten. Abschnitt 3.3.2 widmet sich dann den *bedingt-typbaren* E-Knoten. Auf *nicht-typbare* E-Knoten gehen wir anschließend im Abschnitt 3.3.3 ein.

3.3.1 Echt-typbare Elementknoten

Im Abschnitt 3.3.1.1 stellen wir die im Rahmen der typed-value-orientierten Repräsentation neu eingeführten V-Knoten vor. Anschließend beschäftigt sich Abschnitt 3.3.1.2 mit der Korrespondenz zwischen atomaren Werten und T-Kindknoten. Darauf aufbauend erläutern wir im Abschnitt 3.3.1.3 die Herleitung der typed-value-orientierten Repräsentation. Inwieweit bei dieser Herleitung Informationsverluste auftreten, ist Gegenstand von Abschnitt 3.3.1.4. Im Abschnitt 3.3.1.5 erklären wir dann, warum es bei der typed-value-orientierten Repräsentation nicht notwendig ist, getypte und textuelle Werte in Form von Knoteneigenschaften zu speichern. Abschließend geht es im Abschnitt 3.3.1.6 um die Frage, wie echt-typbare E-Knoten, die in typed-value-orientierter Repräsentation vorliegen, in die traditionelle Repräsentation (rück)überführt werden können.

3.3.1.1 V-Knoten

Die wichtigste Neuerung der typed-value-orientierten Repräsentation besteht darin, dass jeder atomare Wert, der im getypten Wert eines echt-typbaren E-Knotens enthalten ist,

durch einen separaten *Atomarer-Wert-Knoten* (*Atomic Value Node*, kurz *V-Knoten*) repräsentiert wird [MR05]. Bei echt-typbaren E-Knoten wird zudem — um Redundanz zu vermeiden — auf die Darstellung der T-Kindknoten verzichtet [Rab05].

Wir möchten dies kurz an einem Beispiel veranschaulichen und betrachten hierzu den E-Knoten *Jahresliste* aus Abschnitt 3.1.1, dessen getypter Wert eine Folge bestehend aus den drei atomaren Werten *[1976, JahresTyp]*, *[1977, JahresTyp]* und *[2006, JahresTyp]* ist. *Abbildung 3.10* zeigt sowohl die traditionelle als auch die typed-value-orientierte Repräsentation dieses Knotens. Bei der typed-value-orientierten Repräsentation wird jeder der drei atomaren Werte jeweils durch einen eigenen V-Knoten dargestellt — zugleich entfällt der ursprünglich vorhandene T-Knoten *"1976_1977_2006"*.

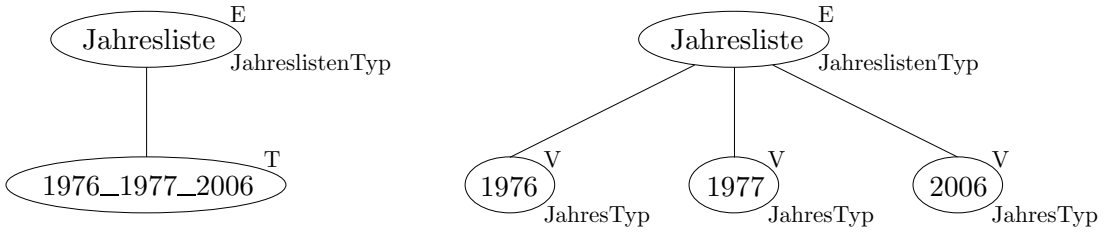


Abbildung 3.10: Repräsentation des E-Knotens *Jahresliste*

Bei den V-Knoten handelt es sich um eine von uns neu eingeführte Knotenart. Jeder V-Knoten repräsentiert einen atomaren Wert. Der wesentliche Unterschied zwischen einem V-Knoten und dem von ihm repräsentierten atomaren Wert besteht darin, dass der V-Knoten (im Gegensatz zum atomaren Wert) einen Vater haben kann.

Wie bei E- und A-Knoten unterscheiden wir auch bei V-Knoten konsequent zwischen der Knotenart (hier: *V*) und dem Knotentyp (z. B. *JahresTyp*). Als Typ eines V-Knotens gelte dabei der Typname des von ihm repräsentierten atomaren Werts. Die graphische Darstellung der V-Knoten erfolgt analog zur Darstellung der atomaren Werte.

Im obigen Beispiel war die Ermittlung der typed-value-orientierten Repräsentation sehr einfach möglich. Dies lag daran, dass der betrachtete E-Knoten (in der traditionellen Repräsentation) nur einen einzigen Kindknoten besessen hat. Wie die typed-value-orientierte Repräsentation eines echt-typbaren E-Knotens im allgemeinen Fall hergeleitet wird, werden wir ausführlich im Abschnitt 3.3.1.3 erläutern. Die dafür notwendigen Vorbetrachtungen führen wir im sich nun anschließenden Abschnitt durch.

3.3.1.2 Korrespondierende T-Kindknoten

Bei der traditionellen Sequenzrepräsentation gibt es eine Korrespondenz zwischen den im getypten Wert eines echt-typbaren E-Knotens auftretenden atomaren Werten und den T-Kindknoten. Wir werden hierauf im Folgenden genauer eingehen.

Es sei an dieser Stelle ausdrücklich darauf hingewiesen, dass sich sämtliche Ausführungen des aktuellen Abschnitts auf die *traditionelle* Sequenzrepräsentation beziehen. Dies gilt insbesondere für Aussagen über T-Kindknoten.

Beim getypten Wert eines echt-typbaren E-Knotens handelt es sich entweder um einen einzelnen atomaren Wert mit einem von *untypedAtomic* abweichenden Typ oder um eine Folge aus mehreren atomaren Werten (Abschnitt 3.3). Damit besitzt ein echt-typbarer E-Knoten einerseits stets die *nilled*-Eigenschaft *false* — andererseits muss er zwangsläufig von einem einfachen Typ oder von einem komplexen Typ mit einfachem Inhalt sein (vgl. Abschnitt 2.10.2.5). Letzteres bedeutet, dass ein echt-typbarer E-Knoten keine E-Kindknoten haben darf (vgl. Abschnitt 2.7.3). Folglich kann ein echt-typbarer E-Knoten keine *indirekten* T-Nachfahrenknoten (wie T-Enkelknoten, T-Urenkelknoten etc.) besitzen, sondern lediglich *direkte* T-Nachfahrenknoten, also T-Kindknoten.

Im Allgemeinen wird der textuelle Wert eines E-Knotens durch die Verkettung der Inhalte aller (direkten und indirekten) T-Nachfahrenknoten gebildet (vgl. Abschnitt 2.10.2.5). Da ein echt-typbarer E-Knoten jedoch keine indirekten T-Nachfahrenknoten haben kann, entspricht sein textueller Wert der Verkettung aller seiner T-*Kind*knoteninhalte. Aus dem entsprechend zusammengesetzten textuellen Wert ergibt sich (in Abhängigkeit vom konkreten Typ des E-Knotens) dann der getypte Wert. Somit korrespondiert jeder im getypten Wert enthaltene atomare Wert mit einem oder mehreren T-Kindknoten.[2] Wir wollen dies anhand eines kleinen Beispiels illustrieren.

Wir betrachten den in *Abbildung 3.11* dargestellten echt-typbaren E-Knoten *gemischteListe*, der diverse T- und C-Kindknoten besitzt. Der E-Knoten sei vom Typ *unserListentyp*, bei welchem es sich um einen Listentyp handle, der als Listeneinträge Werte der Datentypen *JahresTyp* und *boolean* zulässt. Der textuelle Werte des E-Knotens *gemischteListe* lautet *"1976 false 1977 1995 true 2006"*, der getypte Wert *([1976, JahresTyp], [false, boolean], [1977, JahresTyp], [1995, JahresTyp], [true, boolean], [2006, JahresTyp])*.

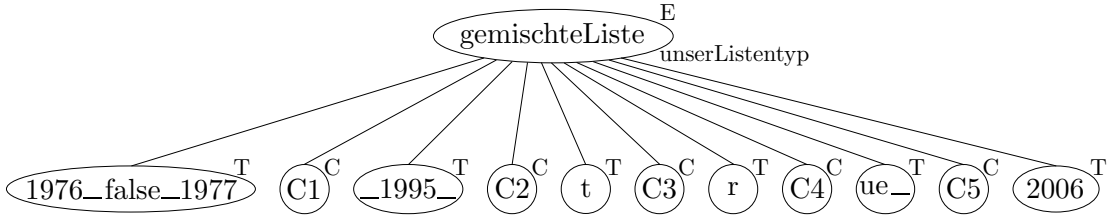


Abbildung 3.11: Traditionelle Repräsentation des E-Knotens *gemischteListe*

In unserem Beispiel korrespondieren die atomaren Werte *[1976, JahresTyp]*, *[false, boolean]* und *[1977, JahresTyp]* mit dem ersten (d. h. linkesten) T-Kindknoten. Der atomare Wert *[1995, JahresTyp]* ist dem zweiten T-Kindknoten zuordenbar, während *[true, boolean]* sowohl mit dem dritten, dem vierten als auch dem fünften T-Kindknoten korrespondiert. Der atomare Wert *[2006, JahresTyp]* kann dem letzten T-Kindknoten zugeordnet werden. *Abbildung 3.12* fasst die Korrespondenz zwischen den atomaren Werten und den T-Kindknoten nochmals zusammen.

[2] Die einzige Ausnahme bildet ein relativ unbedeutender Sonderfall, auf den wir am Ende dieses Abschnitts noch kurz eingehen.

atomarer Wert	*[1976, JahresTyp]*	*[false, boolean]*	*[1977, JahresTyp]*	*[1995, JahresTyp]*	*[true, boolean]*	*[2006, JahresTyp]*
korrespond. T-Kindknoten	"1976␣false␣1977"			"␣1995␣"	"*t*", "*r*" und "*ue*␣"	"2006"

Abbildung 3.12: Korrespondenz zwischen atomaren Werten und T-Kindknoten

Wir möchten nun noch kurz den einzigen Sonderfall vorstellen, bei dem ein atomarer Wert, der im getypten Wert eines echt-typbaren E-Knotens enthalten ist, *nicht* mit einem T-Kindknoten korrespondiert. Dieser Sonderfall liegt vor, wenn ein E-Knoten vom Typ *string* (oder einem darauf basierenden nutzerdefinierten Typ) keinerlei T-Kindknoten besitzt. Sowohl beim textuellen als auch beim getypten Wert dieses E-Knotens handelt es sich dann um die leere Zeichenkette, also die Zeichenkette der Länge 0. Somit enthält der getypte Wert zwar einen atomaren Wert (nämlich die leere Zeichenkette), dieser atomare Wert korrespondiert aber nicht mit einem T-Kindknoten — es existiert ja überhaupt kein T-Kindknoten.

3.3.1.3 Herleitung der typed-value-orientierten Repräsentation

Wir werden im Folgenden erläutern, wie sich die typed-value-orientierte Repräsentation eines echt-typbaren E-Knotens schrittweise ermitteln lässt. Als Ausgangspunkt („0. Schritt") dient dabei die traditionelle Repräsentation des entsprechenden Knotens.

Wir setzen zunächst voraus, dass der am Ende des vorigen Abschnitts angesprochene Sonderfall *nicht* vorliegt. Wir nehmen also einen echt-typbaren E-Knoten an, bei dem jeder im getypten Wert enthaltene atomare Wert mit einem oder mehreren T-Kindknoten korrespondiert.

- *1. Schritt:*

  Jeder atomare Wert wird jeweils dem *linkesten* der mit ihm korrespondierenden T-Kindknoten zugeordnet.

  Bezogen auf den E-Knoten *gemischteListe* aus Abschnitt 3.3.1.2 bedeutet dies, dass der atomare Wert *[true, boolean]* dem T-Kindknoten *"t"* zugeordnet wird, da dies der linkeste der drei mit ihm korrespondierenden T-Kindknoten *"t"*, *"r"* und *"ue␣"* ist. Jeder der restlichen fünf atomaren Werte *[1976, JahresTyp]*, *[false, boolean]*, *[1977, JahresTyp]*, *[1995, JahresTyp]* und *[2006, JahresTyp]* besitzt jeweils nur einen einzigen korrespondierenden T-Kindknoten, welcher damit trivialerweise zugleich der jeweils linkeste der korrespondierenden T-Kindknoten ist. Somit wird jeder dieser fünf atomaren Werte jeweils seinem (einzigen) korrespondierenden T-Kindknoten zugeordnet. *Abbildung 3.13* fasst die Zuordnung der atomaren Werte zu den T-Kindknoten nochmals zusammen.

- *2. Schritt:*

  Die den T-Kindknoten zugeordneten atomaren Werte werden als V-Knoten repräsentiert. Anstelle von zugeordneten atomaren Werten besitzen die T-Kindknoten jetzt zugeordnete V-Knoten *(Abbildung 3.14)*.

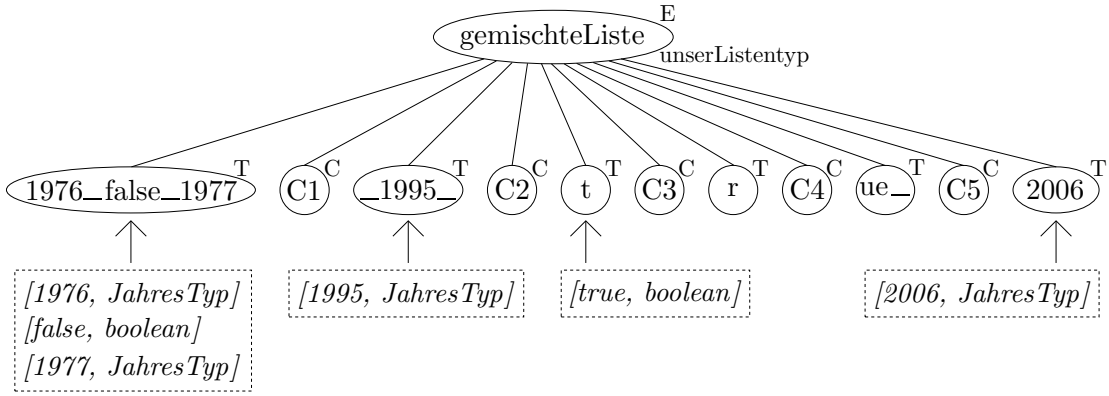


Abbildung 3.13: Zuordnung der atomaren Werte zu den T-Kindknoten

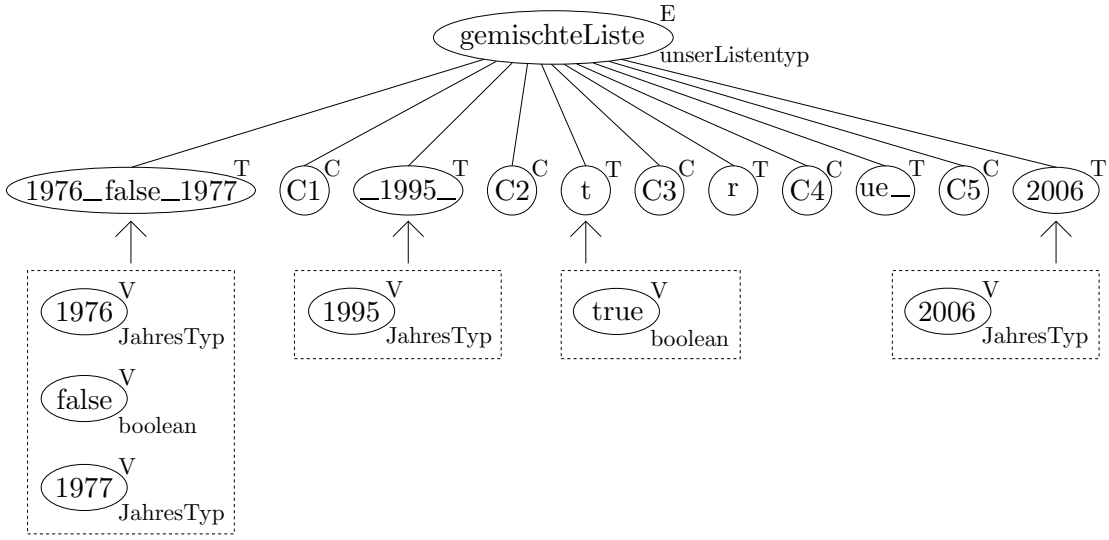


Abbildung 3.14: Zuordnung der V-Knoten zu den T-Kindknoten

- *3. Schritt:*

  Alle T-Kindknoten, die keine zugeordneten V-Knoten haben, werden ersatzlos gestrichen. Jeder der verbleibenden T-Kindknoten wird durch die ihm zugeordneten V-Knoten ersetzt. Sofern ein T-Kindknoten dabei durch *mehrere* V-Knoten ersetzt wird, ergibt sich die Anordnung der V-Knoten aus der Reihenfolge, in der die von den V-Knoten repräsentierten atomaren Werte im getypten Wert des E-Knotens auftreten.

  In unserem Beispiel entfallen die T-Kindknoten *"r"* und *"ue_"* ersatzlos, da sie keine zugeordneten V-Knoten besitzen. Der T-Kindknoten *"1976_false_1977"* wird durch die drei ihm zugeordneten V-Knoten ersetzt. Diese drei V-Knoten repräsentieren die atomaren Werte *[1976, JahresTyp]*, *[false, boolean]* und *[1977, JahresTyp]*, welche in genau dieser Reihenfolge im getypten Wert des E-Knotens auftreten. Somit ergibt sich für die Reihenfolge dieser drei V-Knoten, dass der V-Knoten *[1976, JahresTyp]* als linkester, der V-Knoten *[false, boolean]* als mittlerer und der V-Knoten *[1977,*

JahresTyp] als rechtester der drei Knoten anzuordnen ist. Jeder der restlichen drei T-Kindknoten *"_1995_"*, *"t"* und *"2006"* wird jeweils durch den einen ihm zugeordneten V-Knoten ersetzt. *Abbildung 3.15* zeigt die resultierende typed-value-orientierte Repräsentation des E-Knotens *gemischteListe*. *JTyp* und *bool* dienen dabei (wie auch in einigen noch folgenden Abbildungen) als Kurzschreibweisen für *JahresTyp* bzw. *boolean*.

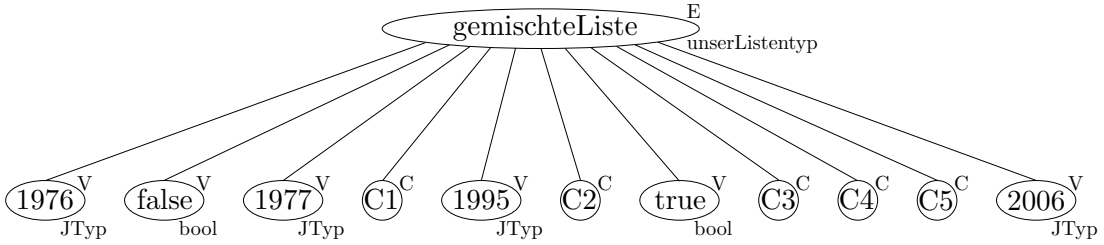


Abbildung 3.15: Typed-value-orientierte Repräsentation des E-Knotens *gemischteListe*

Wir setzen nun das Vorliegen des am Ende des vorigen Abschnitts erwähnten Sonderfalls voraus. Wir nehmen also einen E-Knoten vom Typ *string* (oder einem darauf basierenden Typ) an, der keinerlei T-Kindknoten hat. Ein solcher E-Knoten besitzt als getypten Wert (und auch als textuellen Wert) die leere Zeichenkette. Die Herleitung seiner typed-value-orientierten Repräsentation erfolgt in nur einem Schritt:

- *Herleitungsschritt beim Vorliegen des Sonderfalls:*

  Als linkester Kindknoten wird ein V-Knoten eingefügt, der die leere Zeichenkette repräsentiert.

  Wir möchten dies kurz an einem Beispiel veranschaulichen. Wir betrachten hierzu den im linken Teil der *Abbildung 3.16* (traditionell) dargestellten E-Knoten *Sonderfall*, welcher vom Typ *string* ist und zwar C-Kindknoten, aber keinerlei T-Kindknoten hat. Bei der im rechten Teil der Abbildung gezeigten typed-value-orientierten Repräsentation besitzt dieser E-Knoten einen (neu hinzugekommenen) V-Knoten als linkesten Sohn. Dieser V-Knoten repräsentiert den (einzigen) im getypten Wert des E-Knotens enthaltenen atomaren Wert — nämlich die leere Zeichenkette.

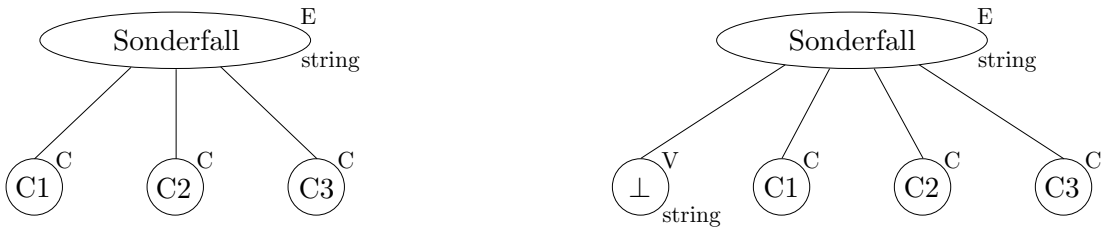


Abbildung 3.16: Repräsentation des E-Knotens *Sonderfall*

In Abbildung 3.16 (und in anderen Abbildungen der vorliegenden Arbeit) verwenden wir $\perp$ als Symbol für die leere Zeichenkette. Damit wollen wir sicherstellen, dass die leere Zeichenkette nicht „übersehen“ wird.

3.3.1.4 Auftretende Informationsverluste

Bei der im vorigen Abschnitt beschriebenen Herleitung der typed-value-orientierten Repräsentation treten u. U. geringfügige Informationsverluste auf. Die Ursache hierfür liegt darin, dass die traditionelle Repräsentation gewisse Informationen enthält, die für die Sequenzcursor-basierte Verarbeitung nicht relevant sind und die deshalb in der typed-value-orientierten Repräsentation nicht widergespiegelt werden. Wir werden hierauf im Folgenden genauer eingehen.

Als Bezugspunkt unserer Erläuterungen dient uns der E-Knoten *kurzeListe*, der (wie der in den vorigen Abschnitten betrachtete E-Knoten *gemischteListe*) vom Typ *unserListentyp* ist. *Abbildung 3.17* zeigt sowohl die traditionelle als auch die typed-value-orientierte Repräsentation dieses Knotens.

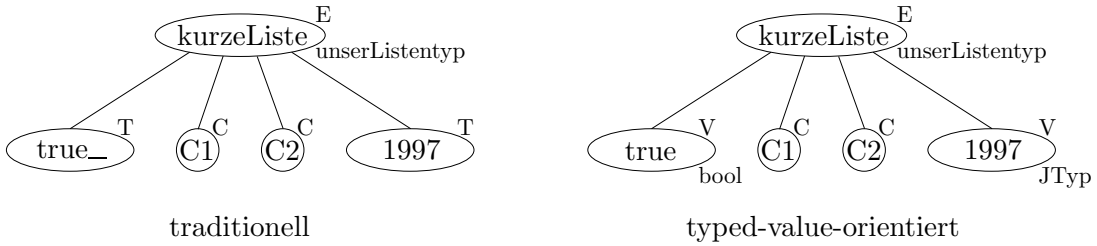


Abbildung 3.17: Repräsentation des E-Knotens *kurzeListe*

Wir werden nun beschreiben, welche (in der traditionellen Repräsentation enthaltenen) Informationen *keinen* Einfluss auf die typed-value-orientierte Repräsentation haben. Für jede dieser Informationen werden wir jeweils auf bestimmte in der *Abbildung 3.18* dargestellte Beispiele verweisen, die zwar bezüglich der entsprechenden Information vom E-Knoten *kurzeListe* aus Abbildung 3.17 abweichen, aber dennoch dieselbe typed-value-orientierte Repräsentation besitzen.

- Leerzeichen dienen bei Listen lediglich als Trennsymbol und haben darüber hinaus keine Bedeutung.
  - Es ist unerheblich, zu welchem T-Kindknoten ein als Trennsymbol genutztes Leerzeichen genau gehört. Die in den Beispielen 1 und 2 der Abbildung 3.18 dargestellten E-Knoten besitzen damit dieselbe typed-value-orientierte Repräsentation wie der E-Knoten aus Abbildung 3.17.
  - Es ist unerheblich, ob zwei aufeinanderfolgende Listeneinträge durch ein oder mehrere Leerzeichen getrennt sind. Folglich haben auch die in den Beispielen 3 und 4 gezeigten E-Knoten dieselbe typed-value-orientierte Repräsentation.
  - Es ist unerheblich, ob links vom ersten Listeneintrag bzw. rechts vom letzten Listeneintrag Leerzeichen existieren oder nicht. Der E-Knoten aus Beispiel 5 besitzt somit ebenfalls die gleiche typed-value-orientierte Repräsentation.

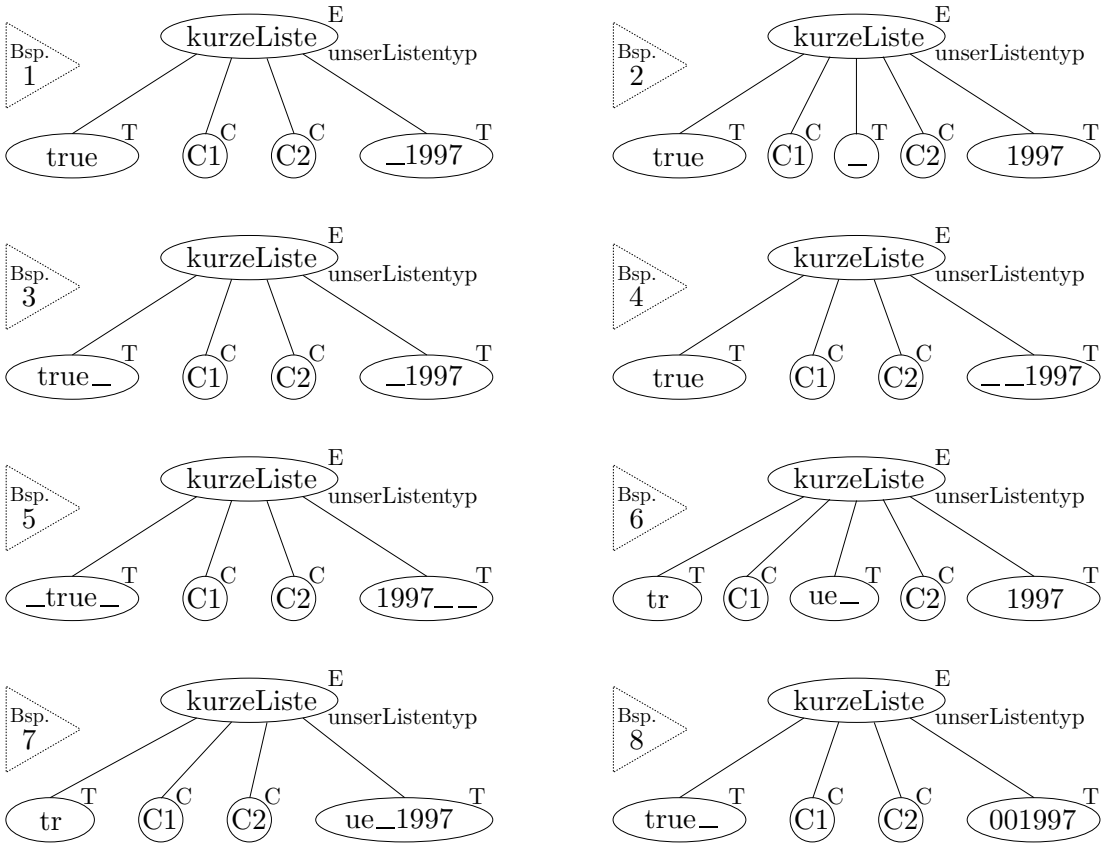


Abbildung 3.18: Vernachlässigbare Abwandlungen des E-Knotens *kurzeListe*

- Es ist unerheblich, ob ein atomarer Wert auf mehrere T-Kindknoten aufgesplittet ist oder nicht. Atomare Werte gelten als *unteilbar* — eine etwaige Aufsplittung ihrer textuellen Repräsentation ist aus semantischer Sicht ohne Bedeutung. Damit besitzen auch die E-Knoten der Beispiele 6 und 7 die im rechten Teil der Abbildung 3.17 gezeigte typed-value-orientierte Repräsentation.
- Es ist unerheblich, welche konkrete textuelle Repräsentation eines atomaren Werts genutzt wird. Bei Integer-Werten ist es beispielsweise nicht relevant, ob die textuelle Repräsentation führende Nullen besitzt. Folglich hat der E-Knoten aus Beispiel 8 ebenfalls dieselbe typed-value-orientierte Repräsentation.

Streng genommen ergibt sich der textuelle Wert eines echt-typbaren E-Knotens aus der Verkettung aller seiner T-Kindknoteninhalte (Abschnitt 3.3.1.2). Somit besitzen die E-Knoten der Beispiele 1, 2, 6 und 7 den textuellen Wert *"true_1997"*, während die E-Knoten der Beispiele 3 und 4 den textuellen Wert *"true__1997"* haben. Der textuelle Wert des E-Knotens aus Beispiel 5 lautet *"_true_1997__"*. Der E-Knoten aus Beispiel 8 besitzt *"true_001997"* als textuellen Wert.

Das XQuery-Datenmodell erlaubt bezüglich des textuellen Werts allerdings ausdrücklich gewisse Freiheitsgrade. Unter anderem ist es irrelevant, ob Listeneinträge durch ein oder mehrere Leerzeichen getrennt sind, ob vor dem ersten bzw. nach dem letzten Listeneintrag Leerzeichen auftreten und ob Zahlenwerte mit führenden Nullen repräsentiert wer-

den. Somit gilt *"true_1997"* als zulässiger textueller Wert für sämtliche in den Abbildungen 3.17 und 3.18 dargestellte E-Knoten. Ein anderer zulässiger textueller Wert wäre z. B. *"_true__01997_"*.

3.3.1.5 Getypter und textueller Wert

Für E-Knoten sieht das XQuery-Datenmodell vor, den getypten Wert und den textuellen Wert in Form von Knoteneigenschaften zu speichern [W3C07e]. Um unnötige Redundanz zu vermeiden, verzichten wir im Rahmen der typed-value-orientierten Repräsentation jedoch auf eine derartige Speicherung [Jat07]. Dies ist möglich, da sich beide Werte sehr leicht aus den V-Kindknoten ermitteln lassen.

Zur Bestimmung des getypten Werts werden einfach alle V-Kindknoten entsprechend ihrer Reihenfolge durchlaufen. *Abbildung 3.19* veranschaulicht dies für den aus den Abschnitten 3.3.1.2 und 3.3.1.3 bekannten E-Knoten *gemischteListe*. Das Durchlaufen der V-Kindknoten entlang der gestrichelten Linie ergibt dabei die Folge *([1976, JahresTyp], [false, boolean], [1977, JahresTyp], [1995, JahresTyp], [true, boolean], [2006, JahresTyp])* — also den getypten Wert des E-Knotens.

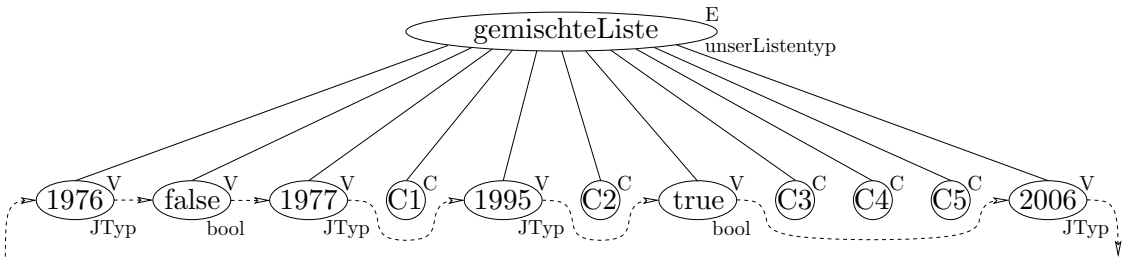


Abbildung 3.19: Ermittlung des getypten Werts des E-Knotens *gemischteListe*

Um den textuellen Wert eines E-Knotens zu ermitteln, werden die von den V-Knoten repräsentierten Werte zunächst in Zeichenketten umgewandelt. Diese Zeichenketten werden anschließend entsprechend der Reihenfolge der V-Knoten verkettet, wobei zwischen aufeinanderfolgenden Zeichenketten jeweils ein Leerzeichen eingefügt wird. *Abbildung 3.20* veranschaulicht dieses Vorgehen am Beispiel des E-Knotens *gemischteListe*. Die sich durch die Verkettung ergebende Zeichenkette lautet *"1976 false 1977 1995 true 2006"*. Es handelt sich dabei um den textuellen Wert des E-Knotens.[3]

3.3.1.6 Rücktransformation

Besitzt ein (in typed-value-orientierter Repräsentation vorliegender) echt-typbarer E-Knoten einen V-Kindknoten, der die leere Zeichenkette repräsentiert, so liegt zwangsläufig der am Ende von Abschnitt 3.3.1.2 beschriebene Sonderfall vor. Um die traditionelle Repräsentation zu ermitteln, ist es dann lediglich erforderlich, diesen V-Kindknoten ersatzlos zu streichen. Existiert hingegen kein V-Kindknoten, der die leere Zeichenkette repräsentiert, so lässt sich die traditionelle Repräsentation wie folgt herleiten:

[3] Genau genommen handelt es sich hierbei um eine von mehreren zulässigen Varianten des textuellen Werts (vgl. Abschnitt 3.3.1.4).

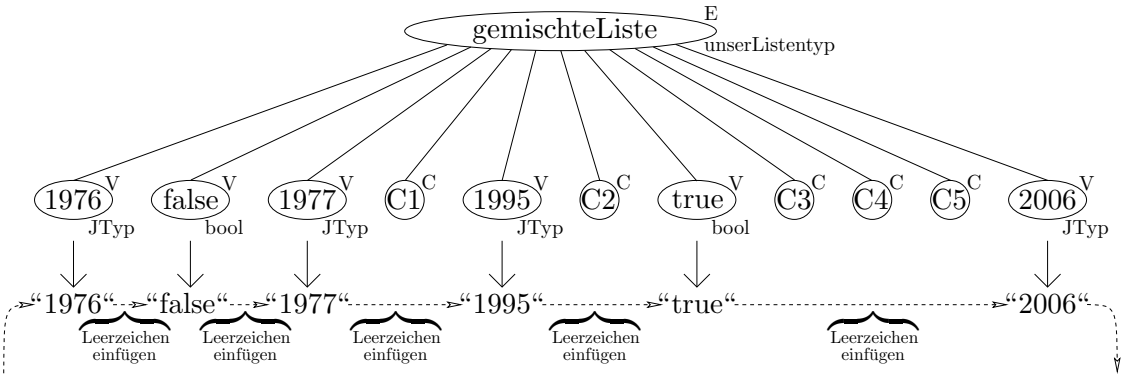


Abbildung 3.20: Ermittlung des textuellen Werts des E-Knotens *gemischteListe*

- *1. Schritt:*

  Zunächst wird der vom linkesten V-Kindknoten repräsentierte Wert in eine Zeichenkette umgewandelt. Anschließend wird der linkeste V-Kindknoten durch einen T-Knoten ersetzt, der genau diese Zeichenkette enthält. Mit den restlichen V-Kindknoten wird analog verfahren, wobei den Zeichenketten jedoch jeweils ein Leerzeichen vorangestellt wird.

  Wird als Ausgangspunkt dieses Schritts die in Abbildung 3.15 dargestellte typed-value-orientierte Repräsentation des E-Knotens *gemischteListe* genutzt, so ergibt sich das in *Abbildung 3.21* gezeigte (Zwischen-)Resultat. Wegen der Existenz unmittelbar benachbarter T-Kindknoten handelt es sich dabei allerdings (noch) *nicht* um eine zulässige Instanz des XQuery-Datenmodells (vgl. Abschnitt 2.10.2.1).

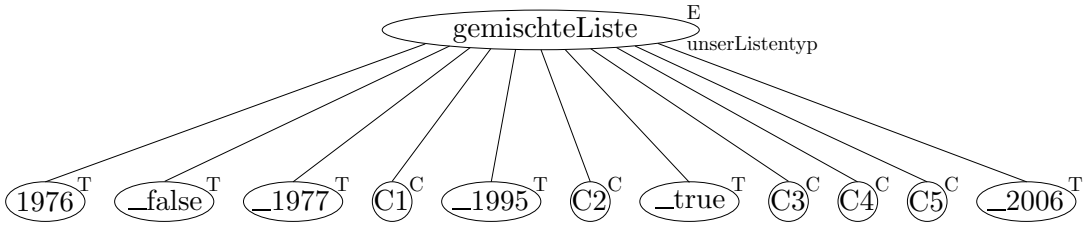


Abbildung 3.21: Zwischenergebnis bei der Rücktransformation

- *2. Schritt:*

  Unmittelbar benachbarte T-Kindknoten werden zu einem T-Kindknoten verschmolzen.

  Für unser Beispiel erhalten wir damit die in *Abbildung 3.22* dargestellte traditionelle Repräsentation des E-Knotens *gemischteListe*. Dass diese Repräsentation nicht mit der in Abbildung 3.11 gezeigten „Originalversion“ identisch ist, liegt an den im Abschnitt 3.3.1.4 beschriebenen Informationsverlusten, die bei der Herleitung der typed-value-orientierten Repräsentation auftreten.

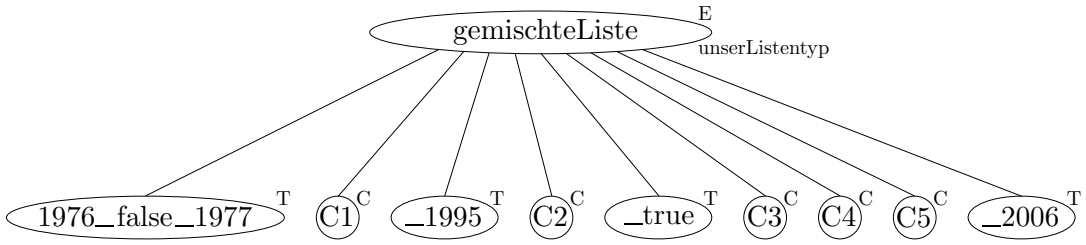


Abbildung 3.22: Resultat der Rücktransformation

3.3.2 Bedingt-typbare Elementknoten

Bei bedingt-typbaren E-Knoten werden sämtliche Kindknoten (d. h. insbesondere auch alle T-Kindknoten) in die typed-value-orientierte Repräsentation übernommen [MR05]. Anders als bei echt-typbaren E-Knoten werden die T-Kindknoten also *nicht* durch V-Knoten ersetzt. Da keine Kindknoten wegfallen, hinzukommen oder ersetzt werden, erübrigt sich hier die Angabe eines Algorithmus zur Herleitung der typed-value-orientierten Repräsentation bzw. zur Rücktransformation.

Wie bei echt-typbaren E-Knoten verzichten wir im Rahmen der typed-value-orientierten Repräsentation auch bei bedingt-typbaren E-Knoten auf die explizite Speicherung von textuellem und getyptem Wert. Dies ist möglich, da sich beide Werte sehr einfach aus den Nachfahrenknoten ermitteln lassen: Der textuelle Wert ergibt sich aus der Verkettung der Inhalte aller T- und V-Nachfahrenknoten.[4] Der getypte Wert entspricht dem textuellen Wert, umgewandelt nach *untypedAtomic* (vgl. Abschnitt 2.10.2.5).

Wir möchten nun noch anhand eines kleinen Beispiels erläutern, wieso es bei bedingt-typbaren E-Knoten *nicht* sinnvoll wäre, die T-Kindknoten durch V-Knoten zu ersetzen. Wir betrachten hierzu den in *Abbildung 3.23* dargestellten E-Knoten *kurzeListe* vom Typ *untyped*. Dieser bedingt-typbare E-Knoten besitzt *"true_1997"* als textuellen Wert, sein getypter Wert lautet *["true_1997", untypedAtomic]*. Wir möchten an dieser Stelle ausdrücklich hervorheben, dass es sich bei diesem getypten Wert *nicht* um eine Folge von zwei atomaren Werten handelt, sondern (wie generell beim getypten Wert eines bedingt-typbaren E-Knotens) um einen *einzelnen* atomaren Wert vom Typ *untypedAtomic*.

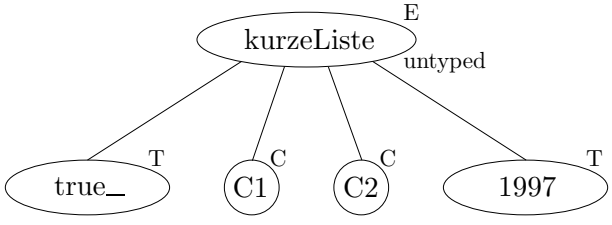


Abbildung 3.23: Bedingt-typbarer E-Knoten *kurzeListe*

[4] Ein bedingt-typbarer E-Knoten kann echt-typbare E-Nachfahrenknoten haben, die ihrerseits V-Kindknoten anstelle von T-Kindknoten besitzen. Aus diesem Grund sind nicht nur T-Nachfahrenknoten, sondern auch V-Nachfahrenknoten zu berücksichtigen. Bei der Verkettung wird zwischen aufeinanderfolgenden V-Knoteninhalten jeweils ein Leerzeichen eingefügt (vgl. Abschnitt 3.3.1.5).

Wollte man die beiden T-Kindknoten nun durch V-Knoten ersetzen, so käme dafür nur ein einziger V-Knoten in Frage — nämlich der V-Knoten, der den atomaren Wert *["true_1997", untypedAtomic]* repräsentiert. Damit ginge jedoch die Information verloren, dass die beiden C-Knoten zwischen *"true_"* und *"1997"* angeordnet sind. Diese Information kann aber für die weitere Verarbeitung relevant sein. Wir werden hierauf im Folgenden kurz eingehen.

Der von uns betrachtete E-Knoten ist vom Typ *untyped*. Dies bedeutet, dass er ein *nicht-validiertes* Element repräsentiert (vgl. Abschnitt 2.10.2.4). Es ist denkbar, dass der E-Knoten später mittels `XMLVALIDATE` validiert wird (vgl. Abschnitt 2.11.3). Dabei könnte ihm beispielsweise der aus dem Abschnitt 3.3.1.2 bekannte Typ *unserListentyp* zugeordnet werden. Der getypte Wert des E-Knotens wäre dann eine Folge aus zwei atomaren Werten. Jeder dieser beiden Werte würde in der typed-value-orientierten Repräsentation jeweils als ein separater V-Knoten dargestellt (vgl. Abbildung 3.17 in Abschnitt 3.3.1.4). Die typed-value-orientierte Repräsentation müsste dabei auch die Information enthalten, dass die beiden C-Knoten zwischen *true* und *1997* positioniert sind. Diese Information ist also wichtig und darf nicht verlorengehen. Um dies zu gewährleisten, werden T-Kindknoten von bedingt-typbaren E-Knoten nicht durch V-Knoten ersetzt.

3.3.3 Nicht-typbare Elementknoten

Bei nicht-typbaren E-Knoten werden (wie bei bedingt-typbaren E-Knoten) alle Kindknoten in die typed-value-orientierte Repräsentation übernommen. Es werden also keine Kindknoten durch V-Knoten ersetzt [Rab05]. Dieses Vorgehen ist sinnvoll, da der getypte Wert eines nicht-typbaren E-Knotens entweder nicht definiert ist oder es sich bei ihm um eine leere Folge handelt. In beiden Fällen existieren keine (im getypten Wert enthaltenen) atomaren Werte, die im Rahmen der typed-value-orientierten Repräsentation mit Hilfe von V-Knoten dargestellt werden könnten. Eine Ersetzung von Kindknoten durch V-Knoten kommt somit gar nicht in Betracht.

Da beim Übergang der traditionellen Repräsentation in die typed-value-orientierte Repräsentation weder Kindknoten entfallen noch dazukommen, erübrigt sich (wie auch bei bedingt-typbaren E-Knoten) die Angabe eines Algorithmus zur Ermittlung der typed-value-orientierten Repräsentation bzw. zur Rücktransformation.

Im Rahmen der typed-value-orientierten Repräsentation wird bei nicht-typbaren E-Knoten auf die explizite Speicherung des textuellen Werts verzichtet. Dieser lässt sich bei Bedarf leicht aus den Nachfahrenknoten ermitteln (vgl. Abschnitt 3.3.2). Bezüglich des getypten Werts wird lediglich die Information gespeichert, ob dieser leer oder nicht definiert ist.

3.4 Repräsentation der getypten Werte von Attributknoten

Der getypte Wert eines A-Knotens ist stets eine Folge aus keinem, einem oder mehreren atomaren Werten (vgl. Abschnitt 2.10.2.5). Bei der typed-value-orientierten Repräsentation wird jeder dieser atomaren Werte als separater V-Knoten dargestellt — und zwar als V-*Kind*knoten des entsprechenden A-Knotens [Yan06].

Wir möchten dies kurz anhand eines Beispiels veranschaulichen und betrachten hierzu den A-Knoten *gemischteListe*. Dieser Knoten besitze den aus Abschnitt 3.3.1.2 bekannten Typ

unserListentyp, sein getypter Wert sei eine Folge, die aus den drei atomaren Werten *[1976, JahresTyp]*, *[false, boolean]* und *[1977, JahresTyp]* besteht. *Abbildung 3.24* zeigt sowohl die traditionelle als auch die typed-value-orientierte Repräsentation dieses Knotens. Bei der typed-value-orientierten Repräsentation wird jeder der drei atomaren Werte durch einen eigenen V-Kindknoten dargestellt. Die Reihenfolge der V-Kindknoten ergibt sich dabei aus der Reihenfolge, in der die atomaren Werte im getypten Wert des A-Knotens auftreten.

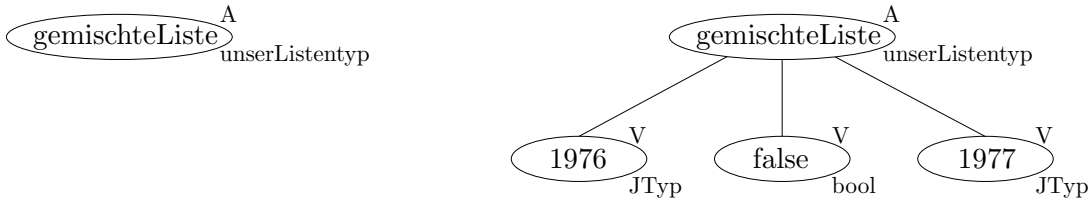


Abbildung 3.24: Repräsentation des A-Knotens *gemischteListe*

Für A-Knoten ist die Herleitung der typed-value-orientierten Repräsentation also deutlich einfacher als für E-Knoten (vgl. Abschnitt 3.3). Dies liegt daran, dass A-Knoten (in der traditionellen Repräsentation) keinerlei Kindknoten besitzen können. Im Gegensatz zu E-Knoten stellt sich für A-Knoten somit gar nicht die Frage, ob T-Kindknoten in die typed-value-orientierte Repräsentation übernommen werden sollten oder wie die V-Kindknoten in Bezug auf etwaige C-Kindknoten anzuordnen sind — es existieren ja weder T- noch C-Kindknoten.

Wie bei E-Knoten sieht das XQuery-Datenmodell auch für A-Knoten vor, den getypten Wert und den textuellen Wert in Form von Knoteneigenschaften zu speichern [W3C07e]. Aus Gründen der Redundanzvermeidung verzichten wir im Rahmen der typed-value-orientierten Repräsentation jedoch auf eine derartige Speicherung [Jat07]. Wie bei echt-typbaren E-Knoten lassen sich beide Werte sehr leicht aus den V-Kindknoten rekonstruieren. Für Details verweisen wir auf den Abschnitt 3.3.1.5.

Ähnlich einfach wie die Ermittlung der typed-value-orientierten Repräsentation ist für A-Knoten auch die Rücktransformation: Um (ausgehend von der typed-value-orientierten Repräsentation) die traditionelle Repräsentation herzuleiten, ist es lediglich erforderlich, zunächst den getypten und den textuellen Wert zu bestimmen und anschließend alle V-Kindknoten ersatzlos zu streichen.

3.5 Repräsentation der getypten Werte anderer Knotenarten

Nachdem wir in den beiden vorangegangenen Abschnitten erläutert haben, wie E- und A-Knoten im Rahmen der typed-value-orientierten Repräsentation dargestellt werden, widmen wir uns nun den D-, P-, C- und T-Knoten. Auf N-Knoten werden wir ausführlich im Abschnitt 3.7 eingehen.

Anders als bei A-Knoten und echt-typbaren E-Knoten werden die getypten Werte von D-, P-, C- und T-Knoten *nicht* mit Hilfe von V-Knoten repräsentiert [Rab05]. Bei der

Herleitung der typed-value-orientierten Repräsentation kommen bei D-, P-, C- und T-Knoten also *keine* neuen Kindknoten hinzu.

Bei D-Knoten werden sämtliche (in der traditionellen Repräsentation vorhandene) Kindknoten in die typed-value-orientierte Repräsentation übernommen. Da P-, C- und T-Knoten im Gegensatz zu D-Knoten keine Kindknoten besitzen können (Abschnitt 2.10.2.2), sind für sie auch keine Kindknoten zu übernehmen.

Bezüglich D-, P-, C- und T-Knoten werden bei der Herleitung der typed-value-orientierten Repräsentation also weder Kindknoten hinzugefügt noch entfernt. Folglich erübrigt sich hier (wie bei bedingt-typbaren und nicht-typbaren E-Knoten) die Angabe eines Transformations- bzw. Rücktransformationsalgorithmus.

Das XQuery-Datenmodell sieht für D-Knoten vor, den getypten Wert und den textuellen Wert als Knoteneigenschaften zu speichern. Zur Vermeidung von Redundanz verzichten wir bei der typed-value-orientierten Repräsentation jedoch auf eine solche Speicherung. Dies ist möglich, da sich beide Werte — wie im Abschnitt 3.3.2 für bedingt-typbare E-Knoten beschrieben — sehr leicht aus den Nachfahrenknoten herleiten lassen. Während wir bei D-Knoten im Rahmen der typed-value-orientierten Repräsentation also ausdrücklich auf die explizite Speicherung von textuellem und getyptem Wert verzichten, sieht das XQuery-Datenmodell für P-, C- und T-Knoten von vornherein gar keine derartige Speicherung vor.

Wir möchten im Folgenden noch kurz darauf eingehen, warum es bei D-, P-, C- und T-Knoten *nicht* sinnvoll wäre, die getypten Werte durch V-Knoten zu repräsentieren:

Der getypte Wert eines D-Knotens ist ein einzelner atomarer Wert vom Typ *untypedAtomic*, der lediglich Informationen enthält, die auch in den T- bzw. V-Nachfahrenknoten des D-Knotens enthalten sind. Würde dieser getypte Wert als eigenständiger V-Knoten dargestellt, würde dadurch unerwünschte Redundanz verursacht. Um dies zu vermeiden, verzichten wir darauf, den getypten Wert eines D-Knotens mittels V-Knoten zu repräsentieren [Rab05].

Die getypten Werte von P-, C- und T-Knoten sind einzelne atomare Werte vom Typ *string* bzw. *untypedAtomic*, die exakt dem Knoteninhalt entsprechen (vgl. Abschnitt 2.10.2.5). Eine Repräsentation dieser getypten Werte als separate V-Knoten hätte somit keinerlei Vorteile. Der Nachteil einer derartigen Repräsentation bestünde darin, dass sich die Knotenanzahl deutlich erhöhen würde. Konsequenterweise wird bei P-, C- und T-Knoten darauf verzichtet, die getypten Werte mit Hilfe von V-Knoten darzustellen.

3.6 Atomare Werte als Sequenzeinträge

Eine traditionell repräsentierte Sequenz entspricht einer (möglicherweise leeren) geordneten Folge aus Bäumen und/oder atomaren Werten (Abschnitt 2.10.3). Als Sequenzeinträge können außer Bäumen also auch atomare Werte auftreten. Im Rahmen der typed-value-orientierten Repräsentation wird jeder dieser atomaren Werte durch einen (vaterlosen) V-Knoten repräsentiert [Jat07]. Diese V-Knoten lassen sich als Bäume auffassen — nämlich als Bäume, die jeweils nur aus einem einzigen Knoten bestehen. Folglich entspricht eine Sequenz in typed-value-orientierter Repräsentation stets einer (möglicherweise leeren) geordneten Folge, die sich *ausschließlich* aus Bäumen zusammensetzt. Im Gegensatz zur

traditionellen Sequenzrepräsentation ist es dabei möglich, dass ein Baum V-Knoten enthält bzw. aus einem einzigen V-Knoten besteht.

Wir möchten dies kurz am Beispiel der Sequenz aus Abschnitt 3.2 veranschaulichen. In traditioneller Repräsentation (Abbildung 3.7) besteht diese Sequenz aus drei Bäumen und einem atomaren Wert. In typed-value-orientierter Repräsentation *(Abbildung 3.25)* setzt sich die Sequenz hingegen aus vier Bäumen zusammen — der ursprünglich vorhandene atomare Wert *[7, integer]* wurde durch einen (aus einem einzigen V-Knoten bestehenden) Baum ersetzt.[5]

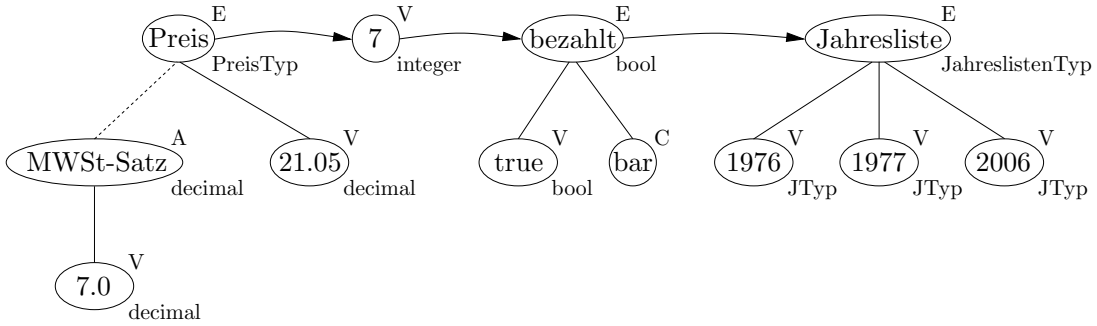


Abbildung 3.25: Beispiel für eine Sequenz in typed-value-orientierter Repräsentation

Während der Herleitung der typed-value-orientierten Repräsentation wird jeder als Sequenzeintrag auftretende atomare Wert durch einen V-Knoten ersetzt. Bei der Rücktransformation ist es somit erforderlich, jeden auf Wurzelebene vorkommenden V-Knoten durch den von ihm repräsentierten atomaren Wert zu ersetzen.

3.7 Umgang mit Namensraumknoten

Die Existenzberechtigung von N-Knoten ist stark umstritten. Wir werden hierauf im Abschnitt 3.7.1 näher eingehen. Anschließend beschäftigt sich Abschnitt 3.7.2 damit, dass im Rahmen der typed-value-orientierten Repräsentation vollständig auf N-Knoten verzichtet wird.

3.7.1 Namensraumknoten und das XQuery-Datenmodell

Wie im Abschnitt 2.7.2 erläutert, wurden XML-Namensräume eingeführt, um Namenskollisionen zu vermeiden. Die Kombination aus Namensraumnamen (z. B. *http://www.meineURL.de/Einkauf*) und lokalem Namen (z. B. *Artikel*) soll dabei die Namenseindeutigkeit garantieren. *[http://www.meineURL.de/Einkauf, Artikel]* ist ein Beispiel für einen entsprechend zusammengesetzten Namen.

In XML-Dokumenten werden anstelle von Namensraumnamen allerdings Namensraumpräfixe als Namensbestandteile verwendet (Abschnitt 2.7.2). Die Namensraumpräfixe (z. B. *thm*) fungieren dabei als Kürzel für die Namensraumnamen. Bezogen auf unser Beispiel

[5] Da bei der graphischen Darstellung nicht zwischen atomaren Werten und V-Knoten unterschieden wird (Abschnitt 3.3.1.1), ist dieser Unterschied bei einem Vergleich der Abbildungen 3.7 und 3.25 allerdings nicht erkennbar.

würde somit *thm:Artikel* als Name genutzt — das Namensraumpräfix *thm* stände hierbei als Kürzel für den Namensraumnamen *http://www.meineURL.de/Einkauf*.

Der Einsatz von Namensraumpräfixen macht es erforderlich, die Information zu verwalten, welcher Namensraumname welchem Namensraumpräfix zugeordnet ist. So müsste in unserem Beispiel die Information vorgehalten werden, dass das Namensraumpräfix *thm* zum Namensraumnamen *http://www.meineURL.de/Einkauf* gehört. Zur Verwaltung dieser Zuordnungsinformationen werden im XML-Infoset (Abschnitt 2.7.4) so genannte *Namensraum-Informationseinheiten* genutzt [W3C04d]. Im XQuery-Datenmodell (Abschnitt 2.10) sollten (so war es zumindest ursprünglich vorgesehen) N-Knoten diese Aufgabe übernehmen.

Jeder N-Knoten repräsentiert eine Bindung zwischen einem Namensraumpräfix und einem Namensraumnamen (z. B. *thm* $\longrightarrow$ *http://www.meineURL.de/Einkauf*). Würde sich das XQuery-Datenmodell auf die Repräsentation wohlgeformter XML-Dokumente (Abschnitt 2.7.1) beschränken, wäre die Nutzung von N-Knoten eine geeignete Möglichkeit, die Auflösbarkeit[6] von Namensraumpräfixen zu gewährleisten. Jeder E-Knoten würde dann für jeden bei ihm vorkommenden Namensraumpräfix jeweils einen zugeordneten N-Knoten besitzen, der dem entsprechenden Namensraumpräfix den passenden Namensraumnamen zuweist. Diese N-Knoten könnten dann zur Auflösung der Namensraumpräfixe herangezogen werden.

Außer der Repräsentation wohlgeformter XML-Dokumente erlaubt das XQuery-Datenmodell u. a. jedoch auch „freistehende“, d. h. vaterlose A-Knoten (Abschnitt 2.10.2.2). Ein solcher A-Knoten kann weder selbst einen zugeordneten N-Knoten besitzen (vgl. Abbildung 2.23 in Abschnitt 2.10.2.2), noch existiert ein Vaterknoten mit zugeordneten N-Knoten — der A-Knoten ist ja vaterlos. Für diese A-Knoten sind somit keinerlei N-Knoten verfügbar. Ein im Namen eines solchen A-Knotens enthaltenes Namensraumpräfix ist folglich nicht mit Hilfe von N-Knoten auflösbar.

Die Nutzung von N-Knoten garantiert also nicht in allen Fällen die Auflösbarkeit von Namensraumpräfixen. Aus diesem Grund wurde beim Entwurf des XQuery-Datenmodells nach einer Möglichkeit gesucht, auf eine von N-Knoten abhängige Auflösung von Namensraumpräfixen verzichten zu können. Der dabei vom W3C gewählte Weg bestand darin, Namen generell als Tripel zu definieren: Entsprechend dem XQuery-Datenmodell besteht jeder Name aus einem optionalen Namensraumpräfix, einem optionalen Namensraumnamen und dem lokalen Namen.[7] Der Namensraumname darf nur fehlen, wenn auch das Namensraumpräfix fehlt. Damit ist keine Auflösung von Namensraumpräfixen mehr erforderlich — der Namensraumname ist (sofern er existiert) ja bereits Bestandteil des Namens.

Wir möchten dies kurz anhand eines Beispiels illustrieren und betrachten hierzu den Namen *[thm, http://www.meineURL.de/Einkauf, Artikel]*. Dieser Name enthält u. a. die Information, dass das Namensraumpräfix *thm* als Kürzel für den Namensraumnamen *http://www.meineURL.de/Einkauf* dient. Eine Auflösung des Namensraumpräfixes *thm* erübrigt sich somit.

Gemäß dem XQuery-Datenmodell ist die Information, welches Namensraumpräfix zu wel-

[6] Unter der *Auflösung* eines Namensraumpräfixes verstehen wir die Ersetzung des als Kürzel dienenden Namensraumpräfixes durch den eigentlichen Namensraumnamen.

[7] Zur Gewährleistung einer besseren Lesbarkeit verzichten wir in dieser Arbeit aber normalerweise auf die Angabe der Namensraumpräfix- und Namensraumnamen-Bestandteile von Namen (vgl. Abschnitt 2.10.1 bzw. 2.10.2.3).

chem Namensraumnamen gehört, also in den Namen selbst enthalten. Die ursprünglich zur Speicherung dieser Information vorgesehenen N-Knoten sind damit überflüssig. Konsequenterweise hätten N-Knoten nicht als Knotenart ins XQuery-Datenmodell aufgenommen werden dürfen. Dies wird auch von namhaften Fachexperten so gesehen. Beispielsweise argumentiert Don Chamberlin in seinem Änderungsvorschlag *No need for namespace nodes* [Cha04] für eine generelle Streichung der N-Knoten. Auch Jim Melton und Stephen Buxton sprechen sich für eine Abschaffung der N-Knoten aus [MB06].

Bedauerlicherweise konnte sich das W3C allerdings nicht dazu durchringen, auf N-Knoten zu verzichten.[8] Infolgedessen wurde eine „Kompromisslösung“ ins XQuery-Datenmodell aufgenommen, die N-Knoten de jure zwar als Knoten „1. Klasse“, de facto jedoch als Knoten „2. Klasse“ behandelt. Die Nichtgleichbehandlung von N-Knoten zeigt sich u. a. in folgenden Festlegungen [W3C07e, W3C07c, W3C07d, W3C07f]:

- Eine auf dem XQuery-Datenmodell basierende Sprache darf die Nutzung von N-Knoten explizit ausschließen.
- Bindungen zwischen Namensraumpräfixen und Namensraumnamen dürfen auf eine beliebige Art und Weise repräsentiert werden. Insbesondere ist es dabei nicht erforderlich, N-Knoten zu verwenden.
- Auf wesentliche Eigenschaften von N-Knoten kann generell nicht zugegriffen werden. Dies betrifft u. a. die Knotenidentität, den (möglichen) Vaterknoten sowie die genaue Knotenposition.
- Das XQuery-Datenmodell sieht eigentlich eine Zugriffsfunktion (Abschnitt 2.10.2.3) vor, die — angewendet auf einen E-Knoten — alle N-Knoten zurückliefert, die diesem E-Knoten zugeordnet sind. Implementierungen des XQuery-Datenmodells wird es jedoch explizit freigestellt, diese Zugriffsfunktion nicht zu implementieren.
- Ein direkter Zugriff auf N-Knoten ist weder in XPath 2.0 noch in XQuery 1.0 (Abschnitt 2.9) möglich.
- Weder XPath 2.0 noch XQuery 1.0 erlaubt die Konstruktion eines vaterlosen N-Knotens.
- Die XQuery Update Facility (Abschnitt 2.9) ermöglicht kein direktes Ändern, Einfügen, Löschen oder Ersetzen von N-Knoten.

Im Rahmen der typed-value-orientierten Repräsentation werden wir komplett auf die Nutzung der umstrittenen N-Knoten verzichten. Detailliertere Erläuterungen hierzu enthält der folgende Abschnitt.

3.7.2 Verzicht auf Namensraumknoten

Bei der Herleitung der typed-value-orientierten Repräsentation werden alle N-Knoten ersatzlos gestrichen. Davon betroffen sind sowohl vaterlose N-Knoten als auch N-Knoten, die einen Vater besitzen. Wir werden hierauf in den Abschnitten 3.7.2.1 bzw. 3.7.2.2 näher eingehen.

[8] Ein Verzicht auf N-Knoten hätte (teils weitreichende) Änderungen an allen Spezifikationen bzw. Spezifikationsentwürfen nach sich gezogen, die auf dem XQuery-Datenmodell aufbauen. Möglicherweise war der diesbezüglich befürchtete Aufwand der ausschlaggebende Grund dafür, die N-Knoten doch beizubehalten.

3.7.2.1 Namensraumknoten ohne Vater

Vaterlose N-Knoten sind lt. XQuery-Datenmodell zwar theoretisch zulässig (Abschnitt 2.10.2.2), in praxisrelevanten Szenarien treten sie aber dennoch nicht auf. Der Grund hierfür besteht darin, dass sich weder mittels XPath 2.0 noch mittels XQuery 1.0 noch mittels XQuery Update Facility noch mittels SQL/XML:2006 vaterlose N-Knoten erzeugen lassen [W3C07c, W3C07d, W3C07f, ISO06]. Außerdem wären vaterlose N-Knoten aus semantischer Sicht sowieso nicht sinnvoll.

In den extrem seltenen und unbedeutenden Fällen, in denen eine (z. B. durch direkte Konstruktion entstandene) Sequenz dennoch vaterlose N-Knoten enthält, entfallen diese Knoten bei der Überführung in die typed-value-orientierte Repräsentation. Bei einer späteren Rücktransformation sind diese N-Knoten dann nicht mehr rekonstruierbar. Der dadurch auftretende Informationsverlust ist aber vollkommen unkritisch, da vaterlose N-Knoten aus semantischer Sicht ohnehin nicht sinnvoll sind und damit auch keine relevanten Informationen enthalten.

3.7.2.2 Namensraumknoten mit Vater

In traditioneller Repräsentation besitzen nicht-vaterlose N-Knoten zwangsläufig einen E-Knoten als Vater (Abschnitt 2.10.2.2). Solche N-Knoten repräsentieren damit entweder die Bindung zwischen einem beim E-Vaterknoten vorkommenden Namensraumpräfix und dem dazugehörigen Namensraumnamen oder die Bindung zwischen dem Namensraumpräfix *xml* und dem Namensraumnamen *http://www.w3.org/XML/1998/namespace*.[9]

Die während der Herleitung der typed-value-orientierten Repräsentation weggefallenen nicht-vaterlosen N-Knoten lassen sich bei einer Rücktransformation somit vollständig rekonstruieren, indem für jeden E-Knoten die beiden folgenden Schritte ausgeführt werden.

- *1. Schritt:*

  Für jedes Namensraumpräfix-Namensraumname-Paar, das beim E-Knoten (in einem Namen) vorkommt, wird jeweils ein entsprechender N-Knoten erzeugt und dem E-Knoten zugeordnet.

- *2. Schritt:*

  Es wird ein N-Knoten erzeugt, der die (Standard-)Bindung zwischen dem Namensraumpräfix *xml* und dem Namensraumnamen *http://www.w3.org/XML/1998/namespace* repräsentiert. Dieser N-Knoten wird ebenfalls dem E-Knoten zugeordnet.

Wir möchten den Umgang mit nicht-vaterlosen N-Knoten noch kurz anhand eines Beispiels illustrieren. Wir betrachten hierzu einen E-Knoten namens *[thm, http://www.meineURL.de/Einkauf, leereListe]*, der vom (leere Listen erlaubenden) Typ *[thm, http://www.meineURL.de/Einkauf, unserListentyp]* ist. *Abbildung 3.26* zeigt die traditionelle Repräsentation dieses Knotens. Dem E-Knoten sind dabei zwei N-Knoten zugeordnet: Der eine der beiden N-Knoten repräsentiert die

[9] Entsprechend dem XQuery-Datenmodell ist jedem E-Knoten u. a. ein N-Knoten zugeordnet, der die (Standard-)Bindung zwischen dem Namensraumpräfix *xml* und dem Namensraumnamen *http://www.w3.org/XML/1998/namespace* repräsentiert [W3C07e].

Bindung zwischen dem Namensraumpräfix *thm* und dem Namensraumnamen *http://www.meineURL.de/Einkauf*, der andere N-Knoten steht für die Standardbindung *xml* ⟶ *http://www.w3.org/XML/1998/namespace*.

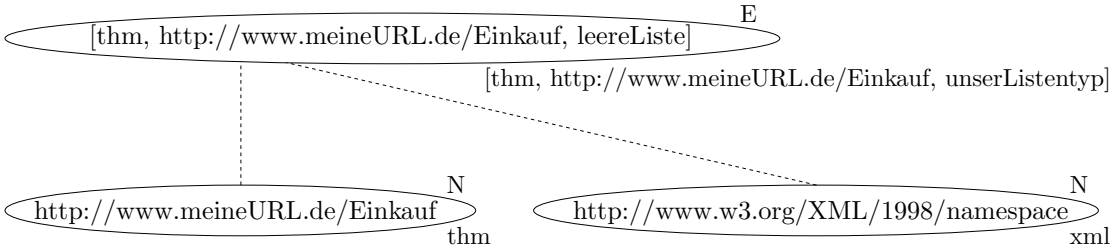


Abbildung 3.26: Traditionelle Repräsentation des E-Knotens *leereListe*

Bei der Überführung in die typed-value-orientierte Repräsentation werden beide N-Knoten ersatzlos getrichen *(Abbildung 3.27)*.

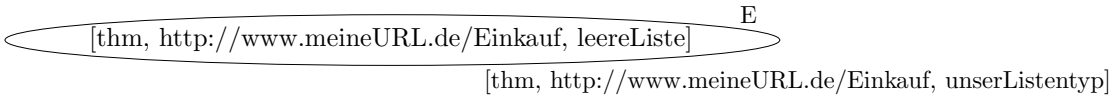


Abbildung 3.27: Typed-value-orientierte Repräsentation des E-Knotens *leereListe*

Soll nun der in typed-value-orientierter Repräsentation vorliegende E-Knoten aus Abbildung 3.27 in die (in Abbildung 3.26 gezeigte) traditionelle Repräsentation rücküberführt werden, so ist dies — wie oben beschrieben — in zwei Schritten möglich: Im ersten Schritt wird das einzige beim E-Knoten vorkommende Namensraumpräfix-Namensraumname-Paar — nämlich *thm* ⟶ *http://www.meineURL.de/Einkauf* — als N-Knoten repräsentiert und dem E-Knoten zugeordnet. Im zweiten Schritt wird ein N-Knoten für die Standardbindung *xml* ⟶ *http://www.w3.org/XML/1998/namespace* erzeugt und ebenfalls dem E-Knoten zugeordnet. Die Reihenfolge der beiden N-Knoten ist dabei unwesentlich (vgl. Abschnitt 2.10.2.1).

3.8 Fazit

Die von uns entwickelte typed-value-orientierte Repräsentation stellt eine neuartige, von der traditionellen Repräsentation abweichende Möglichkeit dar, XQuery-Sequenzen zu repräsentieren. Die wichtigsten Unterschiede gegenüber der traditionellen Repräsentation fassen wir im Abschnitt 3.8.1 nochmals kurz zusammen.

Die typed-value-orientierte Repräsentation wurde speziell an die Bedürfnisse der Sequenzcursor-basierten Verarbeitung (Kapitel 4) angepasst. Damit erfüllt sie (im Wesentlichen) alle im Abschnitt 3.1.2 aufgeführten Anforderungen. Wir werden hierauf im Abschnitt 3.8.2 genauer eingehen.

3.8.1 Unterschiede zur traditionellen Repräsentation

Wir werden im Folgenden kurz diejenigen Punkte betrachten, in denen sich die typed-value-orientierte Repräsentation von der traditionellen Repräsentation unterscheidet. Für nähere Details verweisen wir auf die Ausführungen in den Abschnitten 3.3 bis 3.7.

- *Aufbau einer Sequenz*

  Eine traditionell repräsentierte Sequenz ist eine (möglicherweise leere) Folge aus Bäumen und/oder atomaren Werten. In typed-value-orientierter Repräsentation besteht eine Sequenz hingegen ausschließlich aus Bäumen. Während traditionell repräsentierte Sequenzen also atomare Werte als Sequenzeinträge enthalten können, ist dies bei Sequenzen in typed-value-orientierter Repräsentation nicht möglich.

- *Knotenarten*

  Die traditionelle Sequenzrepräsentation kennt sieben Knotenarten: D-, E-, A-, N-, P-, C- und T-Knoten. Mit Ausnahme der N-Knoten finden sich diese Knotenarten auch bei der typed-value-orientierten Repräsentation wieder. Zusätzlich wird im Rahmen der typed-value-orientierten Repräsentation eine von uns neu eingeführte Knotenart genutzt — die V-Knoten. Die zur Repräsentation von atomaren Werten dienenden V-Knoten können als Kindknoten von E- und A-Knoten auftreten oder vaterlos sein.

  Die typed-value-orientierte Repräsentation unterscheidet somit sieben Knotenarten: D-, E-, A-, P-, C-, T- und V-Knoten. *Abbildung 3.28* veranschaulicht, inwieweit Vater-Kind-Beziehungen zwischen Knoten dieser Knotenarten zulässig sind. Wie in Abbildung 2.23 aus Abschnitt 2.10.2.2 steht ein Haken (✓) dabei für eine zulässige beidseitige Vater-Kind-Beziehung, während ein Stern (∗) eine erlaubte einseitige Vater-Kind-Beziehung symbolisiert. Ein Minuszeichen (−) drückt aus, dass keine Vater-Kind-Beziehung möglich ist.

		Kindknoten						
		D	**E**	**A**	**P**	**C**	**T**	**V**
Vaterknoten	**D**	−	✓	−	✓	✓	✓	−
	E	−	✓	∗	✓	✓	✓	✓
	A	−	−	−	−	−	−	✓

Abbildung 3.28: Mögliche Vater-Kind-Beziehungen zwischen Knoten

- *Speicherung von getyptem und textuellem Wert*

  In traditioneller Repräsentation werden die textuellen und getypten Werte von D-, E- und A-Knoten explizit in Form von Knoteneigenschaften gespeichert. Bei der typed-value-orientierten Repräsentation wird hingegen auf eine derartige Speicherung verzichtet.[10]

[10] Lediglich bei nicht-typbaren E-Knoten wird vermerkt, ob der getypte Wert leer oder nicht definiert ist.

- *Repräsentation der im getypten Wert enthaltenen atomaren Werte*

  Im Gegensatz zur traditionellen Repräsentation wird bei der typed-value-orientierten Repräsentation jeder atomare Wert, der im getypten Wert eines echt-typbaren E-Knotens oder A-Knotens vorkommt, durch einen separaten V-Knoten dargestellt.

3.8.2 Erfüllung der aufgestellten Anforderungen

Im Abschnitt 3.1.2 haben wir verschiedene Anforderungen aufgelistet, die an eine auf die Sequenzcursor-basierte Verarbeitung abgestimmte Sequenzrepräsentation zu stellen sind. Wir werden im Folgenden aufzeigen, dass diese Anforderungen von der typed-value-orientierten Repräsentation (weitestgehend) erfüllt werden.

- *Keine „versteckten" Typinformationen*

  Anstelle von T-Kindknoten ohne spezifische Typinformation besitzen echt-typbare E-Knoten jetzt V-Kindknoten. Jeder dieser V-Kindknoten enthält außer einem Wert auch die dazugehörige Typinformation. Die Typinformation wird hier also nicht im Vaterknoten „versteckt".

- *Vermeidung von Anomalien*

  Um Redundanz (und dadurch hervorgerufene Anomalien) zu vermeiden, werden die textuellen und getypten Werte von D- und A-Knoten nicht als Knoteneigenschaften gespeichert. Gleiches gilt für die getypten Werte von echt-typbaren und bedingt-typbaren E-Knoten sowie für die textuellen Werte sämtlicher E-Knoten.

  Redundanz wird ferner dadurch vermieden, dass die bereits in den Namen enthaltene Information, welcher Namensraumname welchem Namensraumpräfix zugeordnet ist, nicht zusätzlich in Gestalt von N-Knoten gespeichert wird.

- *Keine Aufsplittung atomarer Werte*

  Ein im getypten Wert eines echt-typbaren E-Knotens oder A-Knotens auftretender atomarer Wert wird stets als *einzelner* V-Knoten repräsentiert. Eine Aufsplittung atomarer Werte auf mehrere Knoten ist damit ausgeschlossen.

- *Einfacher Zugriff auf einzelne atomare Werte*

  Jeder atomare Wert, der im getypten Wert eines echt-typbaren E-Knotens oder A-Knotens enthalten ist, wird jeweils durch einen *separaten* V-Knoten dargestellt. Beim Zugriff auf einzelne atomare Werte ist es folglich *nicht* erforderlich, „umständlich" auf irgendwelche Knoten*teile* zuzugreifen. Somit wird also gewährleistet, dass unkompliziert auf die einzelnen atomaren Werte zugegriffen werden kann.

- *Einfache Auswahl einzelner atomarer Werte*

  Jeder V-Knoten repräsentiert jeweils nur genau einen atomaren Wert. Bei der gezielten Auswahl eines einzelnen atomaren Werts ist es somit *nicht* erforderlich, einen Knoten*teil* zu selektieren — stattdessen ist der *komplette* V-Knoten auszuwählen. Die Auswahl einzelner atomarer Werte ist damit unkompliziert möglich.

- *Automatische Transformierbarkeit*

  Wir haben in den Abschnitten 3.3 bis 3.7 ausführlich beschrieben, wie sich die

typed-value-orientierte Repräsentation aus der traditionellen Repräsentation herleiten lässt. Darüber hinaus haben wir detailliert geschildert, wie eine Rücktransformation durchzuführen ist. Basierend auf diesen Beschreibungen kann problemlos ein Programm entwickelt werden, das sowohl die Herleitung der typed-value-orientierten Repräsentation als auch die Rücktransformation automatisch durchführen kann.

Mit dem von uns entwickelten Prototyp (Kapitel 8) konnten wir zeigen, dass die Ermittlung der typed-value-orientierten Repräsentation — ebenso wie die Rücktransformation — tatsächlich automatisch möglich ist.

- *Äquivalenz des Informationsgehalts*

  Da bei der Herleitung der typed-value-orientierten Repräsentation u. U. geringfügige Informationsverluste auftreten, ist keine hundertprozentige Äquivalenz des Informationsgehalts gewährleistet. Dies ist jedoch vollkommen unkritisch, da lediglich unbedeutende Informationen betroffen sind.

  Von entscheidender Wichtigkeit ist die Tatsache, dass die Sequenzen bei der Überführung in die typed-value-orientierte Repräsentation *nicht* mit zusätzlichen Informationen angereichert werden. Es findet also *keine* Erweiterung des Informationsgehalts statt. Informationen, die entsprechend dem XQuery-Datenmodell ohnehin Bestandteil einer (traditionell repräsentierten) Sequenz sind, werden im Rahmen der typed-value-orientierten Repräsentation lediglich auf eine andere Art und Weise dargestellt.

3.9 Abschließendes Beispiel

Im Abschnitt 2.13 haben wir das Beispielszenario *Kundenkartenverwaltung* eingeführt. Jedes Tupel der dabei betrachteten *Kukabo*-Tabelle (Abbildung 2.30) enthält jeweils eine Bonusangebotesequenz. Auf die typed-value-orientierte Repräsentation der in diesen Sequenzen enthaltenen Bonusangebote werden wir im Folgenden näher eingehen.

Ein Bonusangebot wird innerhalb einer Bonusangebotesequenz durch einen einzelnen Baum repräsentiert. Bei der Wurzel eines solchen (exemplarisch in *Abbildung 3.29* dargestellten) Baumes handelt es sich um einen E-Knoten namens *BonusAngebot*, der vom Typ *BonusAngebotsTyp* (kurz *BATyp*) ist. Gemäß seinem Typ sind diesem E-Knoten die beiden A-Knoten *UNr* und *Gültigkeit* zugeordnet, ferner besitzt er die E-Kindknoten *SachAngebot* und *GegenWert*.

Der A-Knoten *UNr* (Kurzform von *Unternehmensnummer*) ist vom Typ *integer* (kurz *int*) und besitzt einen V-Kindknoten, der die Nummer desjenigen Unternehmens enthält, bei dem das Bonusangebot erworben wurde.

Der A-Knoten *Gültigkeit* vom Typ *GültigkeitsTyp* (kurz *GkTyp*) hat drei oder mehr V-Kindknoten vom Typ *date*. Der erste dieser V-Kindknoten gibt an, ab wann das Bonusangebot gilt. Das Datum, an dem das Bonusangebot abläuft, wird durch den zweiten V-Kindknoten dargestellt. Der dritte V-Kindknoten enthält das Datum, an dem das Bonusangebot erworben wurde. Die (möglicherweise vorhandenen) weiteren V-Kindknoten geben Auskunft darüber, wann nachträgliche Änderungen am Bonusangebot vorgenommen wurden. Bei solchen Änderungen kann es sich z. B. um Aufwertungen des Bonusangebots handeln (vgl. Abschnitt 2.13). Der vierte V-Kindknoten repräsentiert dabei das

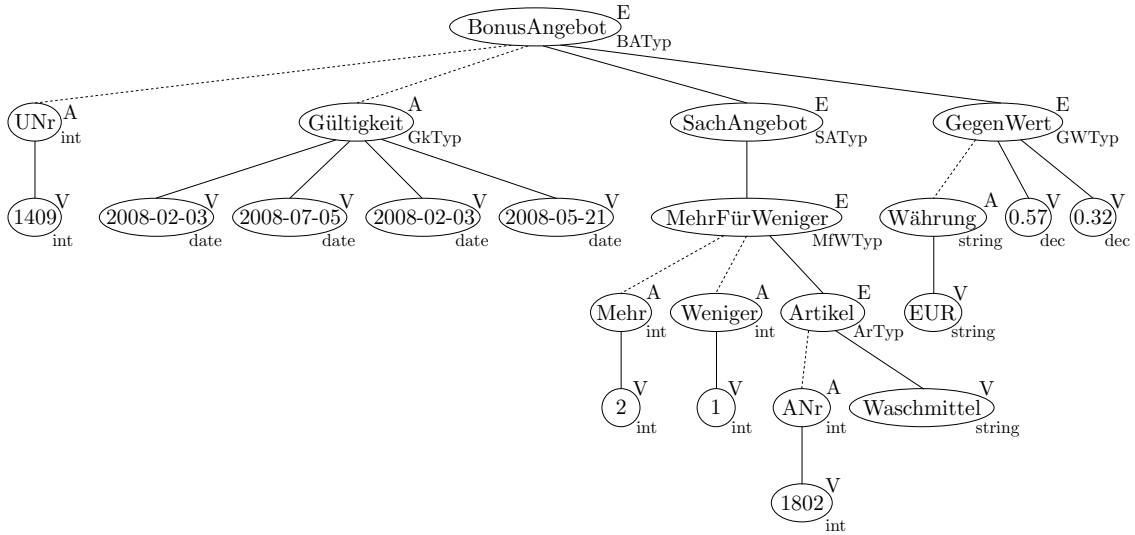


Abbildung 3.29: Bonusangebot zum vergünstigten Erwerb von Waschmittel

Datum der ersten Änderung, der fünfte V-Kindknoten steht für das Datum der zweiten Änderung usw.

Der E-Knoten *SachAngebot* besitzt den Typ *SachAngebotsTyp* (kurz *SATyp*). Bei seinem Sohn kann es sich beispielsweise um den E-Knoten *MehrFürWeniger* (siehe Abbildung 3.29) oder den E-Knoten *Rabatt* (siehe *Abbildung 3.30*) handeln.

Ein Teilbaum mit dem E-Wurzelknoten *MehrFürWeniger* steht für ein Angebot der Art *„x Stück eines Artikels kaufen, aber nur für y Stück bezahlen“* (mit z. B. $x = 2$ und $y = 1$). Dem E-Knoten *MehrFürWeniger* vom Typ *MehrFürWenigerTyp* (kurz *MfWTyp*) sind dabei die A-Knoten *Mehr* und *Weniger* des Typs *integer* zugeordnet. Beide A-Knoten haben jeweils einen V-Kindknoten. Diese V-Kindknoten geben an, wieviele Exemplare des Artikels zum Preis von wievielen Exemplaren gekauft werden können. Auf welchen Artikel sich das Angebot bezieht, geht aus dem Teilbaum mit dem E-Wurzelknoten *Artikel* hervor. Der Wurzelknoten dieses Teilbaums (also der E-Knoten *Artikel*) ist dabei ein Kindknoten vom E-Knoten *MehrFürWeniger*.

Der E-Knoten *Artikel* ist vom Typ *ArtikelTyp* (kurz *ArTyp*) und besitzt einen V-Kindknoten des Typs *string*, der die Artikelbezeichnung enthält. Darüber hinaus ist dem E-Knoten ein A-Knoten namens *ANr* (Kurzform von *Artikelnummer*) zugeordnet. Dieser A-Knoten vom Typ *integer* hat einen V-Kindknoten, der für die Artikelnummer steht.

Ein Teilbaum mit dem E-Wurzelknoten *Rabatt* (vgl. Abbildung 3.30) repräsentiert ein Rabattangebot der Art *„Beim Kauf eines bestimmten Artikels wird ein Preisnachlass von x Prozent gewährt“* (mit z. B. $x = 25.0$). Der E-Knoten *Rabatt* vom Typ *RabattTyp* (kurz *RaTyp*) besitzt dabei einen E-Kindknoten namens *Artikel*. Hierdurch ist festgelegt, für welchen Artikel der Preisnachlass gilt. Ferner ist dem E-Knoten *Rabatt* der A-Knoten *Prozent* zugeordnet. Dieser A-Knoten des Typs *decimal* (kurz *dec*) hat einen V-Kindknoten, der angibt, wie hoch der gewährte prozentuale Preisnachlass ist.

Der E-Knoten *GegenWert* ist vom Typ *GegenWertsTyp* (kurz *GWTyp*) und besitzt zwei V-Kindknoten des Typs *decimal*. Diese V-Kindknoten repräsentieren den einlösbaren Gegenwert bei einem Folgeeinkauf im gleichen Geschäft bzw. bei einem Folgeeinkauf in einem

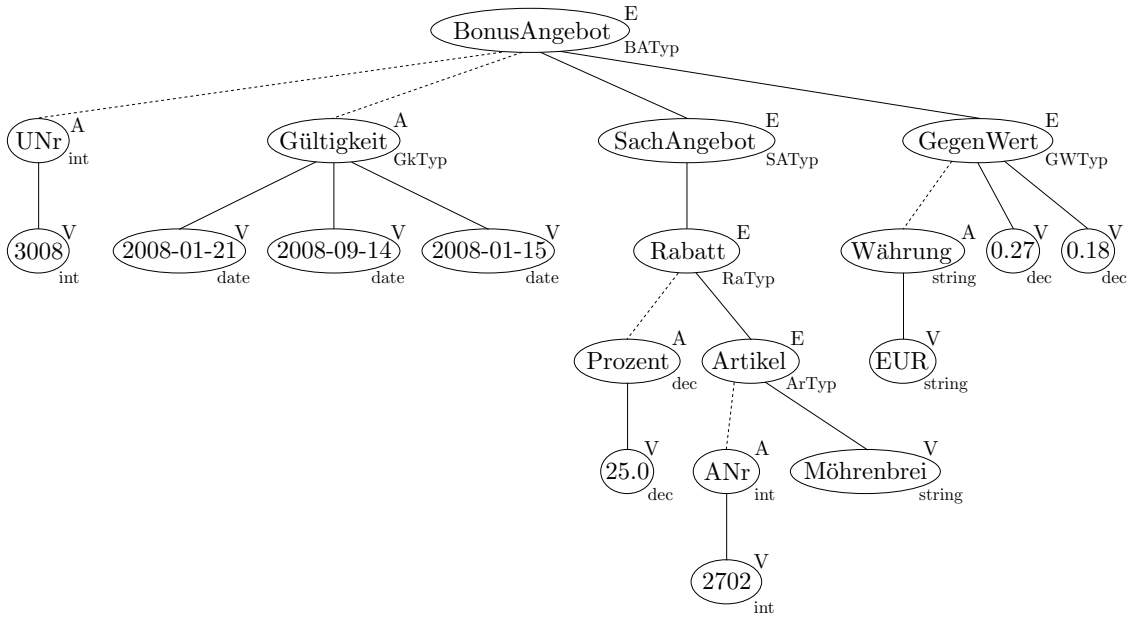


Abbildung 3.30: Bonusangebot zum vergünstigten Erwerb von Möhrenbrei

anderen Geschäft. Dem E-Knoten ist zudem der A-Knoten *Währung* des Typs *string* zugeordnet. Der V-Kindknoten dieses A-Knotens gibt an, in welcher Währung die einlösbaren Gegenwerte vorliegen.

Der in Abbildung 3.29 gezeigte Baum steht also für ein am 03.02.2008 im Geschäft mit der Unternehmensnummer 1409 erworbenes Bonusangebot, welches vom 03.02.2008 bis zum 05.07.2008 gilt. Das Bonusangebot, welches einmalig am 21.05.2008 geändert wurde, ermöglicht den Kauf von zwei Päckchen des Waschmittels mit der Artikelnummer 1802 zum Preis von einem Päckchen. Der einlösbare Gegenwert dieses Bonusangebots beträgt 57 Cent bzw. 32 Cent — je nachdem, ob der Gegenwert im selben oder in einem anderen Geschäft in Anspruch genommen wird.

Das Bonusangebot aus Abbildung 3.30 gilt im Zeitraum vom 21.01.2008 bis 14.09.2008. Es wurde am 15.01.2008 im Geschäft mit der Unternehmensnummer 3008 erworben und seitdem nicht geändert. Beim Kauf von Möhrenbrei mit der Artikelnummer 2702 wird ein Preisnachlass in Höhe von 25 Prozent gewährt. Der einlösbare Gegenwert beträgt 27 Cent bzw. 18 Cent.

Kapitel 4

Sequenzcursor-basierte Verarbeitung

Gemäß SQL/XML:2006 (seit 2006 Teil der aktuellen SQL-Norm) können Anfrageergebnisse beliebige (XQuery-)Sequenzen enthalten (vgl. Abschnitt 2.11.3). Damit stellt sich die Frage, wie solche (in SQL-Anfrageergebnissen enthaltenen) Sequenzen von Anwendungsprogrammen verarbeitet werden können. Eine Antwort hierauf liefert der von uns konzipierte Ansatz der *Sequenzcursor-basierten Verarbeitung* [Mül05, KM06, Mül06], den wir in diesem Kapitel vorstellen.

Nach einer Beschreibung des prinzipiellen Ablaufs der Sequenzcursor-basierten Verarbeitung im Abschnitt 4.1 gehen wir im Abschnitt 4.2 näher auf die einzelnen Verarbeitungsschritte ein. Anschließend beschäftigt sich Abschnitt 4.3 mit möglichen Erweiterungen des Verarbeitungsablaufs. Denkbare Alternativen zur Sequenzcursor-basierten Verarbeitung sind Gegenstand von Abschnitt 4.4. Abschließend erfolgt im Abschnitt 4.5 eine Abgrenzung gegen bereits existierende Verarbeitungsansätze.

4.1 Verarbeitungsablauf

Ausgangspunkt der Sequenzcursor-basierten Verarbeitung ist eine Tabelle mit einer oder mehreren Spalten des SQL/XML:2006-Basisdatentyps `XML`. Diese Tabelle darf zusätzlich auch Spalten anderer Datentypen besitzen. Mit Hilfe einer gegen diese Tabelle gestellten SQL-Anfrage wird ein Anfrageergebnis ermittelt, welches ebenfalls eine oder mehrere XML-Spalten (sowie möglicherweise weitere Nicht-XML-Spalten) umfasst. Als Spaltenwerte der Ergebnistupel treten damit Sequenzen (sowie gegebenenfalls Werte anderer Datentypen) auf. Sofern für die Ergebnisspalten (SQL-)Nullwerte möglich sind, können als Spaltenwerte auch Nullwerte vorkommen.

Mittels „herkömmlichen“ SQL-Tupelcursors wird nun „klassisch“ auf dem Anfrageergebnis navigiert (vgl. Abschnitt 2.4). Der Umgang mit (evtl.) im Anfrageergebnis enthaltenen Nullwerten erfolgt dabei „traditionell“, wie im Abschnitt 2.4.1 beschrieben. Während die Spaltenwerte der (möglicherweise vorhandenen) Nicht-XML-Ergebnisspalten auf herkömmliche Art und Weise verarbeitet werden (vgl. Abschnitt 2.4), erfahren die Spaltenwerte der XML-Ergebnisspalten eine gesonderte Behandlung. Wir werden hierauf im Folgenden näher eingehen.

Beim Weiterbewegen des Tupelcursors (SQL-`FETCH`-Anweisung) werden Spaltenwerte, bei denen es sich um Sequenzen handelt, *nicht* automatisch ins Anwendungsprogramm übertragen.[1] Stattdessen werden — während der Tupelcursor auf dem aktuell zu verarbeitenden Ergebnistupel positioniert ist — Ausschnitte der im entsprechenden Ergebnistupel enthaltenen (möglicherweise sehr großen) Sequenzen definiert. Die Definition dieser *Sequenzausschnitte* geschieht dabei mit Hilfe so genannter *Sequenzcursor* — einer von uns neu eingeführten Cursorart. Die Sequenzcursor werden hierzu zunächst innerhalb der Sequenzen geeignet positioniert und anschließend werden die Sequenzausschnitte mit Bezugnahme auf die Sequenzcursor definiert. Die vom Tupelcursor im Sinne einer Vater-Kind-Beziehung abhängigen Sequenzcursor erlauben es dem Anwendungsprogramm also, in die im Anfrageergebnis enthaltenen XQuery-Sequenzen „einzutauchen“, um Sequenzausschnitte auszuwählen.

Die definierten Sequenzausschnitte werden (während der Tupelcursor immer noch auf dem entsprechenden Ergebnistupel positioniert ist) ins Anwendungsprogramm übertragen, um dort für eine lokale Verarbeitung zur Verfügung zu stehen. Denkbar ist dabei einerseits ein rein lesender Zugriff auf die Sequenzausschnitte und andererseits die Möglichkeit, Änderungen an den Sequenzausschnitten vorzunehmen. Im letzteren Fall soll vom Anwendungsprogramm entschieden werden können, ob die vorgenommenen Änderungen in die Datenbank (wieder)eingebracht werden sollen. Voraussetzung hierfür ist natürlich, dass ein Einbringen der Änderungen überhaupt möglich ist. Wir werden hierauf im Abschnitt 6.6.2 näher eingehen.

Nachdem das lokale Arbeiten auf den ins Anwendungsprogramm übertragenen Sequenzausschnitten abgeschlossen ist (und die gegebenenfalls lokal vorgenommenen Änderungen eingebracht bzw. verworfen wurden), wird der Tupelcursor weiterbewegt, um das nächste Ergebnistupel zu verarbeiten. Nach Abarbeitung der Ergebnistupel können die (nun nicht mehr benötigten) Sequenzcursor gelöscht und der Tupelcursor geschlossen werden.

Abbildung 4.1 fasst den zuvor beschriebenen prinzipiellen Ablauf der Sequenzcursor-basierten Verarbeitung nochmals kompakt zusammen. Um eine bessere Übersichtlichkeit zu gewährleisten, wurden dabei Arbeitsschritte, die lediglich der Behandlung von Nullwerten oder der Verarbeitung von Nicht-XML-Werten dienen, nicht explizit dargestellt.

Die graphische Veranschaulichung des Verarbeitungsablaufs setzt ein *änderndes* lokales Arbeiten voraus. Soll lokal hingegen nur *rein lesend* auf die Sequenzausschnitte zugegriffen werden, entfällt der Schritt *„Einbringen/Verwerfen der lokalen Änderungen“*.

Der gezeigte Ablauf geht ferner davon aus, dass alle Sequenzcursor bereits vor dem ersten (Weiter-)Bewegen des Tupelcursors erzeugt und erst nach dem letzten (Weiter-)Bewegen des Tupelcursors gelöscht werden. Möglich wäre alternativ aber auch eine Erzeugung bzw. Löschung der Sequenzcursor zu anderen Zeitpunkten. So könnten die einzelnen Sequenzcursor beispielsweise erst unmittelbar vor ihrer ersten Positionierung erzeugt und gleich im Anschluss an ihre letzte Nutzung gelöscht werden. Auf weitere denkbare Abwandlungen und „Abkürzungen“ des dargestellten Verarbeitungsablaufs werden wir im Abschnitt 4.3 eingehen.

Eine wesentliche Eigenschaft der Sequenzcursor-basierten Verarbeitung besteht darin, dass

[1] Dieses (an das im Abschnitt 2.5 vorgestellte LOB-Locator-Prinzip angelehnte) Vorgehen weicht vom traditionellen SQL-Cursorkonzept ab: Wird ein Tupelcursor mittels der SQL-`FETCH`-Anweisung weiterbewegt, werden dadurch normalerweise *sämtliche* Spaltenwerte des nun aktuellen Ergebnistupels ins Anwendungsprogramm übertragen.

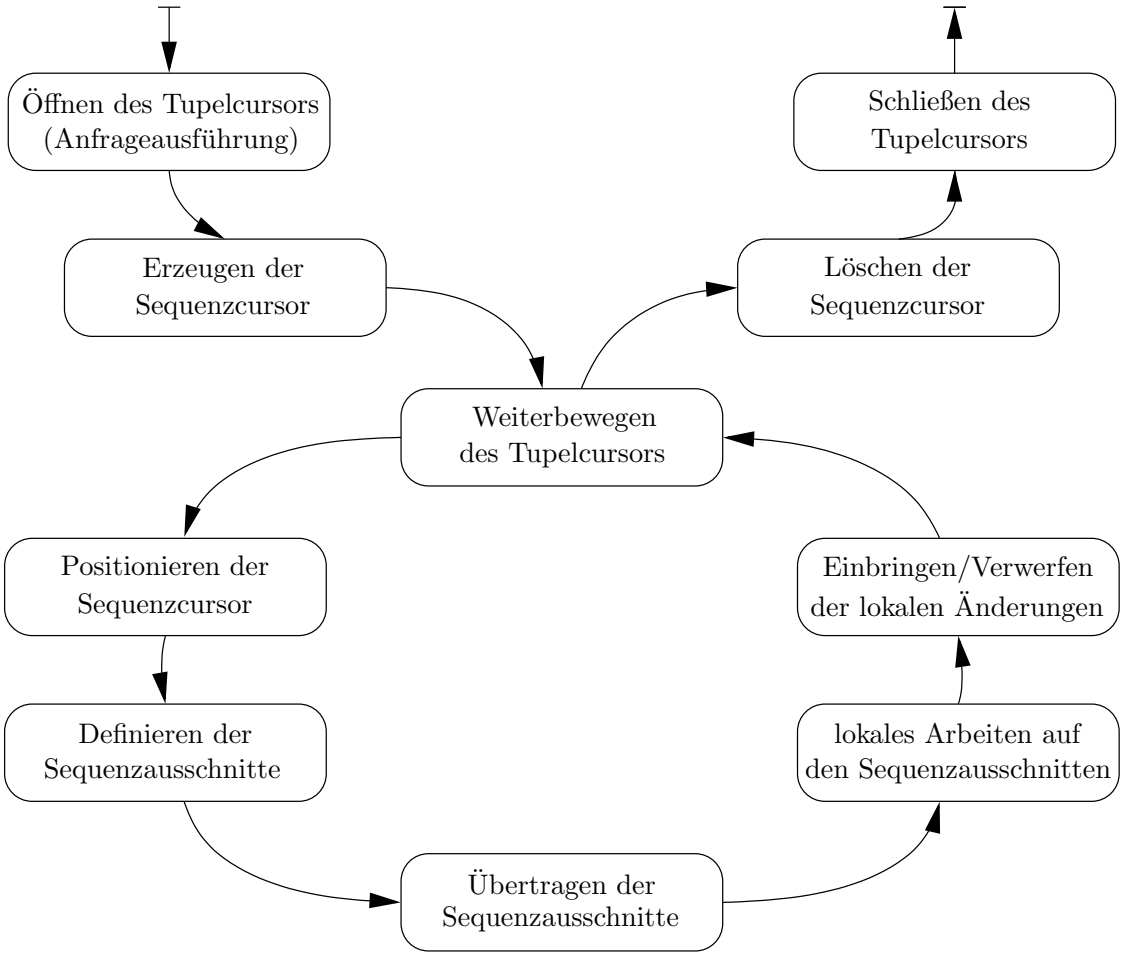


Abbildung 4.1: Sequenzcursor-basierter Verarbeitungsablauf

sie *vollständig* auf der im Kapitel 3 eingeführten typed-value-orientierten Repräsentation basiert.[2] Dies bedeutet, dass sowohl die Positionierung der Sequenzcursor als auch die Definition der Sequenzausschnitte auf der Grundlage der typed-value-orientierten Sequenzrepräsentation erfolgt. Gleiches gilt für das lokale Arbeiten auf den Sequenzausschnitten.

Der Umstand, dass die Sequenzcursor-basierte Verarbeitung auf der typed-value-orientierten Repräsentation beruht, bedeutet übrigens nicht zwangsläufig, dass die in den Ergebnistupeln enthaltenen Sequenzen bereits in typed-value-orientierter Repräsentation vorliegen müssen. Sofern die im Anfrageergebnis vorkommenden Sequenzen traditionell repräsentiert sein sollten, können diese (in einem Zwischenschritt) *automatisch* in die typed-value-orientierte Repräsentation überführt werden (vgl. Abschnitt 3.8.2). Ein entsprechendes Vorgehen wird beispielsweise bei den Prototyparchitekturvarianten genutzt, die in den Abschnitten 8.1.2 bis 8.1.4 vorgestellt werden.

[2] Im Abschnitt 3.1 hatten wir ausführlich erläutert, warum die traditionelle Sequenzrepräsentation *nicht* als Grundlage für die Sequenzcursor-basierte Verarbeitung geeignet ist. Daraufhin hatten wir eigens eine neuartige, speziell auf die Sequenzcursor-basierte Verarbeitung abgestimmte Repräsentationsvariante eingeführt — die typed-value-orientierte Repräsentation.

4.2 Verarbeitungsschritte

Im Folgenden werden wir die einzelnen Arbeitsschritte der Sequenzcursor-basierten Verarbeitung anhand des im Abschnitt 2.13 eingeführten Beispielszenarios *Kundenkartenverwaltung* näher erläutern. Wir nehmen dabei an, dass der Kunde mit der Kundennummer *4711* an der Kundenbefragung teilgenommen hat und somit jeweils die letzten fünf Bonusangebote derjenigen Kategorien aufwerten darf, in denen er innerhalb der letzten drei Monate aktiv war.

Die in den (sich hier anschließenden) Abschnitten 4.2.1 bis 4.2.9 enthaltenen Erklärungen der einzelnen Arbeitsschritte beschränken sich im Wesentlichen auf die für das konkrete Beispiel relevanten Aspekte. Sämtliche darüber hinausgehende Details sind in die beiden folgenden Kapitel ausgelagert. So werden Sequenzcursor ausführlich im Kapitel 5 behandelt, während sich Kapitel 6 intensiv mit Sequenzausschnitten beschäftigt.

In einigen der Arbeitsschritte werden so genannte *Sequenzcursoranweisungen* genutzt. Dabei handelt es sich um von uns neu eingeführte Anweisungen zur Unterstützung der Sequenzcursor-basierten Verarbeitung. Diese, an die Syntax der Datenbanksprache SQL angelehnten Anweisungen wurden von uns als (mögliche) Erweiterung von SQL konzipiert. Wir werden die einzelnen Sequenzcursoranweisungen detailliert im Kapitel 7 vorstellen.

4.2.1 Festlegen der SELECT-Anfrage und Anfrageausführung

In unserem Beispiel dient die Kunden-Kategorie-Bonus-Tabelle *Kukabo* aus Abbildung 2.30 als Ausgangspunkt der Sequenzcursor-basierten Verarbeitung. In dieser Tabelle sind die Bonusangebote jedes Kunden jeweils getrennt nach der Kategorie (z. B. Dienstleistung, Einkauf, Anmietung) abgelegt (vgl. Abschnitt 2.13).

Mit Hilfe der in *Abbildung 4.2* gezeigten SQL-SELECT-Anfrage werden nun diejenigen Bonusangebotesequenzen ausgewählt, die die aufwertbaren Bonusangebote enthalten. Es handelt sich dabei um jene Bonusangebotesequenzen des Kunden mit der Kundennummer *4711*, die zu einer Kategorie gehören, in der der entsprechende Kunde innerhalb der letzten drei Monate aktiv war. Zu jeder dieser Bonusangebotesequenzen wird auch die dazugehörige Kategorie(bezeichnung) ermittelt.

```
SELECT Kategorie, Bonus
FROM   Kukabo
WHERE  KdNr = 4711
   AND letztmals_aktiv >= CURRENT DATE - 3 MONTHS;
```

Abbildung 4.2: Auswahl der relevanten Bonusangebotesequenzen

Wir setzen im Folgenden voraus, dass die SELECT-Anfrage an einen Tupelcursor namens *unserTC* gebunden ist. Für Details bezüglich der Zuordnung einer SELECT-Anfrage zu einem Tupelcursor verweisen wir auf unsere Ausführungen in den Abschnitten 2.4.1 und 2.4.2.

Durch das Öffnen des Tupelcursors *(Abbildung 4.3)* wird die Ausführung der SELECT-Anfrage veranlasst. Der Tupelcursor *unserTC* steht anschließend — wie bei der SQL-Cursorverarbeitung üblich — *vor* (und nicht *auf*) dem ersten Ergebnistupel. Wenn wir

den 02.08.2008 als Tag der Anfrageausführung unterstellen, enthält das Anfrageergebnis *(Abbildung 4.4)* insgesamt drei Ergebnistupel.[3]

```
OPEN unserTC;
```

Abbildung 4.3: Veranlassung der Anfrageausführung

Resultat	Kategorie	Bonus
	Dienstleistung	
	Einkauf	
	Flugbuchung	

Abbildung 4.4: Anfrageergebnis

4.2.2 Erzeugen der Sequenzcursor

Im konkreten Beispiel werden zwei Sequenzcursor benötigt. Beim Definieren eines Sequenzausschnitts (Abschnitt 4.2.5) werden diese Sequenzcursor genutzt, um den Anfang (d. h. den linken Rand) bzw. das Ende (d. h. den rechten Rand) des Sequenzausschnitts festzulegen.

Das Erzeugen der zwei von uns genutzten Sequenzcursor *meinSC1* und *meinSC2* erfolgt mit Hilfe der in *Abbildung 4.5* gezeigten Sequenzcursoranweisung. Die beiden Sequenzcursor werden dabei an die Ergebnisspalte *Bonus* des mit Hilfe des Tupelcursors *unserTC* erstellten Anfrageergebnisses gebunden. Für nähere Details bezüglich der Sequenzcursoranweisung `CREATE SEQUENCE CURSORS` verweisen wir auf Abschnitt 7.2.1.

```
CREATE SEQUENCE CURSORS meinSC1, meinSC2
FOR unserTC
ON RESULT COLUMN Bonus;
```

Abbildung 4.5: Erzeugen der benötigten Sequenzcursor

4.2.3 Weiterbewegen des Tupelcursors

Das Weiterbewegen des Tupelcursors erfolgt wie in Abschnitt 2.4 beschrieben mit Hilfe der SQL-`FETCH`-Anweisung. Obwohl dabei normalerweise sämtliche Spaltenwerte des nun aktuellen Ergebnistupels ins Anwendungsprogramm übertragen werden, soll in unserem Beispiel *keine* (komplette bzw. automatische) Übertragung des Spaltenwerts der XML-Ergebnisspalte *Bonus* stattfinden. Um dies dem SQL-Laufzeitsystem zu signalisieren, wird

[3] In Abbildung 4.4 haben wir auf die Darstellung des Tupelcursors verzichtet. Die im Anfrageergebnis enthaltenen Sequenzen sind stark vereinfacht dargestellt.

für die Hostvariable, die den betreffenden Spaltenwert eigentlich aufnehmen würde, der spezielle (von uns neu eingeführte) Hostvariablentyp `SQL TYPE IS XML SCBP` vereinbart. Das Kürzel *SCBP* steht dabei für *sequence cursor based processing*.

Bei `SQL TYPE IS XML SCBP` handelt es sich (wie bei dem im Abschnitt 2.5 erwähnten Hostvariablentyp `SQL TYPE IS CLOB AS LOCATOR`) nicht um einen Datentyp im herkömmlichen Sinne. Stattdessen dient dieser Datentyp dazu, dem SQL-Laufzeitsystem mitzuteilen, dass der betreffende Spaltenwert beim Ausführen der SQL-`FETCH`-Anweisung *nicht* ins Anwendungsprogramm zu übertragen ist. Die Kennzeichnung der Sequenzcursor-basiert zu verarbeitenden Spaltenwerte erfolgt also vollkommen analog zur Kennzeichnung der mittels des LOB-Locator-Prinzips zu verarbeitenden Spaltenwerte (vgl. Abschnitt 2.5).

Im Fall von *statischem* Embedded SQL wird der Tupelcursor mittels der in *Abbildung 4.6* dargestellten Anweisung weiterbewegt. Die Hostvariable *bonus* besitzt dabei den Typ `SQL TYPE IS XML SCBP`. Die Festlegung dieses Typs erfolgt innerhalb der *embedded SQL declare section* (vgl. Abschnitt 2.3.3).

```
FETCH unserTC INTO :kategorie, :bonus;
```

Abbildung 4.6: Weiterbewegen des Tupelcursors (statisches ESQL)

Bei *dynamischem* Embedded SQL lässt sich der Tupelcursor mit Hilfe der in *Abbildung 4.7* gezeigten Anweisung weiterbewegen. Die Variable *pSqlda* dient dabei als Zeiger auf die SQLDA (vgl. Abschnitt 2.4.2). Vor der Ausführung der SQL-`FETCH`-Anweisung muss in der SQLDA vermerkt werden, dass die der zweiten Ergebnisspalte zugeordnete Hostvariable vom Typ `SQL TYPE IS XML SCBP` ist.

```
FETCH unserTC USING DESCRIPTOR :*pSqlda;
```

Abbildung 4.7: Weiterbewegen des Tupelcursors (dynamisches ESQL)

Wir haben hier grundsätzlich unterstellt, dass die in der XML-Ergebnisspalte *Bonus* enthaltenen Sequenzen gemäß unseres Sequenzcursor-basierten Verarbeitungsablaufs verarbeitet werden sollen. Alternativ ist es jedoch auch möglich, auf die Sequenzcursor-basierte Verarbeitung zu verzichten und die XML-Werte *komplett* (und serialisiert) ins Anwendungsprogramm zu übertragen (vgl. Abschnitt 4.5.1). Die Entscheidung zwischen beiden Varianten wird bei der Festlegung der Hostvariablentypen getroffen: Hostvariablen vom Typ `SQL TYPE IS XML SCBP` signalisieren dem SQL-Laufzeitsystem eine Sequenzcursor-basierte Verarbeitung, während Hostvariablen vom (von der SQL-Norm bereitgestellten) Typ `SQL TYPE IS XML AS CLOB` den kompletten serialisierten XML-Wert aufnehmen.

Der `FETCH`-Anweisung kann man damit nicht direkt ansehen, ob Sequenzcursor-basiert gearbeitet wird oder ob die kompletten serialisierten XML-Werte übertragen werden. Dies ist vollkommen analog dazu, dass der `FETCH`-Anweisung ebenfalls nicht angesehen werden kann, ob LOB-Locatoren eingesetzt werden oder ob eine Übertragung der kompletten LOB-Werte stattfindet (vgl. Abschnitt 2.5).

Sofern das Anfrageergebnis mehrere XML-Spalten besitzt, ist prinzipiell auch eine gemeinsame Nutzung der beiden Verarbeitungsansätze denkbar: Die XML-Werte einer Ergebnisspalte könnten Sequenzcursor-basiert verarbeitet werden, während die XML-Werte

einer anderen Ergebnisspalte komplett und serialisiert ins Anwendungsprogramm übertragen werden. Wir wollen im Folgenden jedoch voraussetzen, dass die Spaltenwerte von XML-Ergebnisspalten *stets* Sequenzcursor-basiert verarbeitet werden.

4.2.4 Positionieren der Sequenzcursor

Wir nehmen an, dass der Tupelcursor *unserTC* aktuell auf dem ersten Ergebnistupel positioniert ist. Wir nutzen nun die beiden Sequenzcursor *meinSC1* und *meinSC2*, um in die in diesem Ergebnistupel enthaltene Bonusangebotesequenz „einzutauchen“. Mittels der in *Abbildung 4.8* dargestellten Sequenzcursoranweisungen bewegen wir zunächst beide Sequenzcursor auf die Wurzel des letzten (also rechtesten) Baums[4] *(Abbildung 4.9)*. Die hierbei genutzte Sequenzcursoranweisung `MOVE SEQUENCE CURSOR` wird ausführlich im Abschnitt 7.2.3 vorgestellt.

```
MOVE meinSC1 TO ROOT OF LAST TREE;
MOVE meinSC2 TO ROOT OF LAST TREE;
```

Abbildung 4.8: Positionieren der Sequenzcursor — Schritt 1

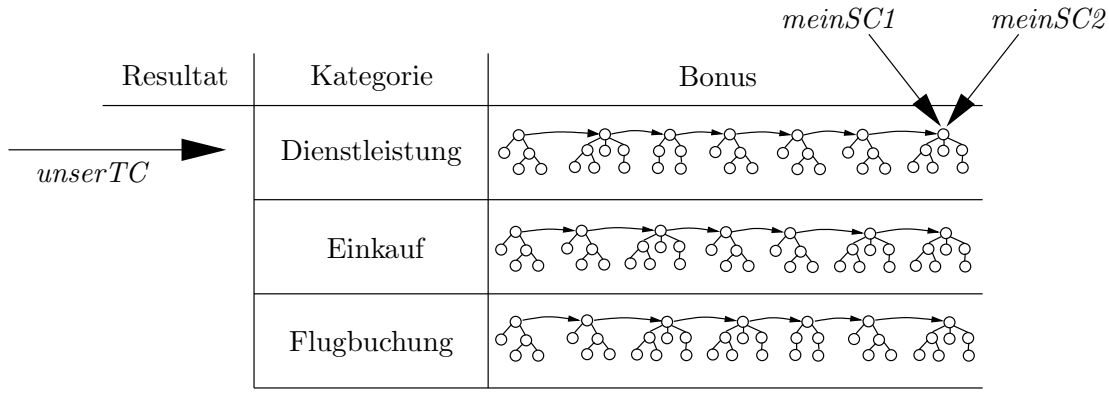


Abbildung 4.9: Anfrageergebnis mit Cursorn (nach Schritt 1)

Mit Hilfe der in *Abbildung 4.10* gezeigten Sequenzcursoranweisung wird der Sequenzcursor *meinSC1* jetzt um vier Schritte nach links bewegt. Der Sequenzcursor *meinSC1* steht anschließend auf der Wurzel des fünftletzten Baums, während der Sequenzcursor *meinSC2* immer noch auf die Wurzel des letzten Baums positioniert ist *(Abbildung 4.11)*.

```
MOVE meinSC1 BACKWARD 4 STEPS;
```

Abbildung 4.10: Positionieren der Sequenzcursor — Schritt 2

[4]Die Positionierung der Sequenzcursor erfolgt auf der Basis der typed-value-orientierten Sequenzrepräsentation (vgl. Abschnitt 4.1). Da sich eine typed-value-orientierte Sequenz ausschließlich aus Bäumen zusammensetzt (Abschnitt 3.8.1), verwenden wir hier den Begriff *Baum* anstelle des (allgemeineren) Begriffs *Sequenzeintrag*.

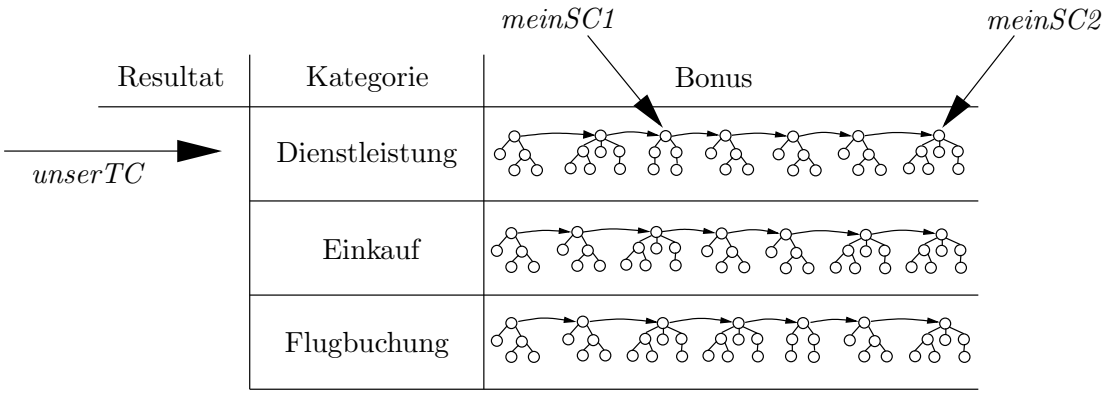


Abbildung 4.11: Anfrageergebnis mit Cursorn (nach Schritt 2)

Der letzte Baum der im aktuellen Ergebnistupel enthaltenen Sequenz entspricht dem zuletzt erworbenen Bonusangebot der Kategorie *Dienstleistung* (vgl. Abschnitt 2.13). Der zweitletzte Baum repräsentiert das vorletzte in dieser Kategorie erworbene Bonusangebot usw. Die Sequenzcursor *meinSC1* und *meinSC2* zeigen also auf das fünftletzte bzw. das letzte Bonusangebot der Kategorie *Dienstleistung*. Die beiden Sequenzcursor markieren somit das linkeste und das rechteste der fünf aufwertbaren Bonusangebote dieser Kategorie.

4.2.5 Definieren der Sequenzausschnitte

Mit Bezugnahme auf die Sequenzcursor *meinSC1* und *meinSC2* lässt sich der gewünschte Sequenzausschnitt wie in *Abbildung 4.12* dargestellt definieren. Die beiden Sequenzcursor werden dabei zur Festlegung der linken bzw. rechten Begrenzung des Sequenzausschnitts genutzt. Für nähere Details bezüglich der hier verwendeten Sequenzcursoranweisung `DEFINE SEQUENCE PART` verweisen wir auf Abschnitt 7.3.1.

```
DEFINE SEQUENCE PART AS FROM meinSC1 TO meinSC2;
```

Abbildung 4.12: Definieren des Sequenzausschnitts

Der soeben definierte Sequenzausschnitt umfasst die letzten fünf Bäume der Bonusangebotesequenz des ersten Ergebnistupels und damit genau die aufwertbaren Bonusangebote der Kategorie *Dienstleistung*. *Abbildung 4.13* zeigt die Position des Tupelcursors, der beiden Sequenzcursor sowie den definierten Sequenzausschnitt.

Sequenzausschnitte werden bei ihrer Definition nicht benannt [Böh07]. Die spätere Bezugnahme auf einen Sequenzausschnitt erfolgt über die Kombination aus Ergebnisspaltenname (hier: *Bonus*) und Tupelcursorname (hier: *unserTC*). Dies ist möglich, da pro Sequenz jeweils nur ein Sequenzausschnitt erlaubt ist und Sequenzausschnitte nur im aktuellen Ergebnistupel existieren dürfen. Wir werden hierauf im Abschnitt 6.1.5 genauer eingehen.

Bei Bedarf (z. B. wenn — wie in SQL prinzipiell zulässig — mehrere gleichnamige Ergebnisspalten existieren) kann bei der Bezugnahme auf einen Sequenzausschnitt anstelle des

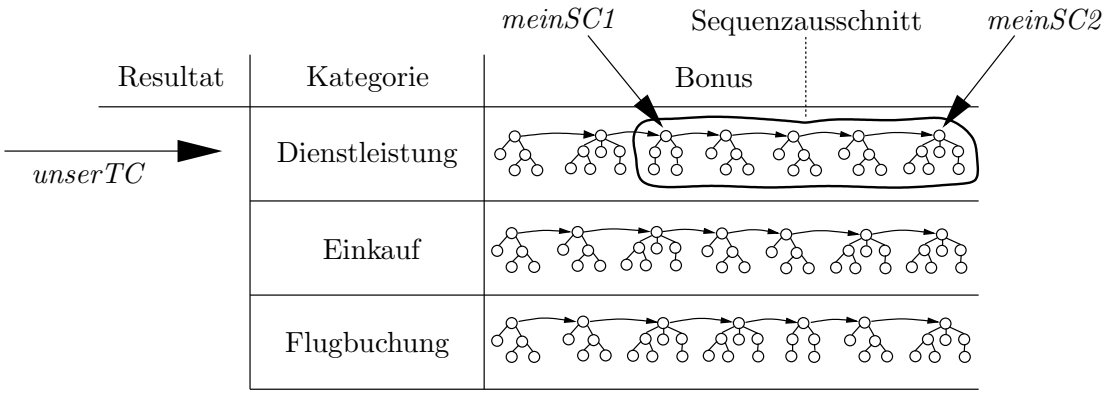


Abbildung 4.13: Anfrageergebnis mit Cursorn und Sequenzausschnitt

Ergebnisspaltennamens auch die Ergebnisspaltennummer genutzt werden. Der Sequenzausschnitt wird dann durch die Kombination aus Ergebnisspaltennummer (hier: *2*) und Tupelcursorname identifiziert.

Ein Sequenzausschnitt braucht (im Sinne der Graphentheorie) *nicht* zusammenhängend zu sein — er kann also aus mehreren (nicht miteinander verbundenen) Teilen bestehen [Böh07]. Diese Teile, die für jeden Sequenzausschnitt jeweils mit 1 beginnend durchnummeriert sind, werden im Folgenden als *Sequenzausschnittsteile* bezeichnet. Für weitere Details verweisen wir auf Abschnitt 6.1.4. Da der von uns definierte Sequenzausschnitt zusammenhängend ist, besteht er aus nur einem einzigen Sequenzausschnittsteil — dem Sequenzausschnittsteil *1*.

4.2.6 Übertragen der Sequenzausschnitte

Mit Hilfe der in *Abbildung 4.14* gezeigten Anweisung wird zunächst ermittelt, wie groß ein lokaler Speicherbereich sein muss, um den kompletten Sequenzausschnitt aufnehmen zu können.[5] Der entsprechende Sequenzausschnitt wird dabei durch die Kombination aus Ergebnisspaltenname und Tupelcursorname identifiziert. Nähere Informationen zur hier genutzten Sequenzcursoranweisung enthält Abschnitt 7.3.3.

```
DESCRIBE SEQUENCE PART
CONTAINED IN RESULT COLUMN Bonus
OF unserTC;
```

Abbildung 4.14: Ermittlung des lokal benötigten Speicherplatzes

Bei der Ausführung der Anweisung `DESCRIBE SEQUENCE PART` vermerkt das SQL-Laufzeitsystem die Größe des lokal benötigten Speicherbereichs in der Komponente *size_of_required_storage* der *SQLSPCA (SQL Sequence Part Communications Area)*. Bei der von uns konzipierten SQLSPCA [Jat07] handelt es sich (wie bei der im Abschnitt 2.4.2

[5] Die (Mindest-)Größe des entsprechenden Speicherbereichs hängt natürlich davon ab, in welcher Art und Weise der Sequenzausschnitt lokal repräsentiert wird. Wir werden im Abschnitt 6.4.1 näher auf die lokale Repräsentation der Sequenzausschnitte eingehen.

betrachteten SQLDA) um eine Datenstruktur im Anwendungsprogramm, die zur Kommunikation mit dem SQL-Laufzeitsystem genutzt wird. Die SQLSPCA wird detailliert im Abschnitt 7.4.2 vorgestellt.

Das Anwendungsprogramm nutzt die in der SQLSPCA abgelegte Größenangabe, um (wie in *Abbildung 4.15* dargestellt) ein Objekt der von uns bereitgestellten Klasse *SPM (sequence part manager)* zu erzeugen [Ott07]. Beim Erzeugen dieses Objekts wird (automatisch) ein lokaler Speicherbereich entsprechender Größe allokiert, der dafür vorgesehen ist, den zu übertragenden Sequenzausschnitt aufzunehmen.

```
pSPM = new SPM(sqlspca.size_of_required_storage);
```

Abbildung 4.15: Erzeugen eines Objekts zur lokalen Sequenzausschnittsverwaltung

Das erzeugte Objekt (welches durch die Zeigervariable *pSPM* referenziert wird) dient zur lokalen Verwaltung des Sequenzausschnitts. Es bietet verschiedene Methoden an, mit deren Hilfe (lokal) lesend und ändernd auf dem Sequenzausschnitt gearbeitet werden kann. Wir werden hierauf im nächsten Abschnitt genauer eingehen.

Unter Zuhilfenahme der SQLSPCA teilt das Anwendungsprogramm nun dem SQL-Laufzeitsystem die Adresse des (zur Aufnahme des Sequenzausschnitts vorgesehenen) Speicherbereichs mit [Böh07]. Hierzu vermerkt das Anwendungsprogramm die entsprechende Adresse in der SQLSPCA-Komponente *pointer_to_local_storage (Abbildung 4.16)*. Wir unterstellen hierbei, dass die Adresse des allokierten Speicherbereichs über die Methode *get_pointer_to_allocated_storage()* abgefragt werden kann.

```
sqlspca.pointer_to_local_storage =
  pSPM->get_pointer_to_allocated_storage();
```

Abbildung 4.16: Übergabe der Adresse des lokalen Speicherbereichs

Die (eigentliche) Übertragung des Sequenzausschnitts (auf die wir im Abschnitt 6.4 genauer eingehen werden) wird durch die in *Abbildung 4.17* gezeigte Anweisung veranlasst. Die Bezugnahme auf den zu übertragenden Sequenzausschnitt erfolgt dabei (wieder) über die Kombination aus Ergebnisspaltenname und Tupelcursorname. Für nähere Details bezüglich der verwendeten Sequenzcursoranweisung verweisen wir auf Abschnitt 7.3.4.

```
TRANSFER SEQUENCE PART
CONTAINED IN RESULT COLUMN Bonus
OF unserTC;
```

Abbildung 4.17: Übertragen des Sequenzausschnitts

Beim Ausführen der Anweisung `TRANSFER SEQUENCE PART` greift das SQL-Laufzeitsystem zunächst auf die Komponente *pointer_to_local_storage* der SQLSPCA zu, um die Adresse des zur Aufnahme des Sequenzausschnitts vorgesehenen Speicherbereichs auszulesen. Anschließend überträgt das SQL-Laufzeitsystem den Sequenzausschnitt in diesen (lokalen) Speicherbereich.

4.2.7 Lokales Arbeiten auf den Sequenzausschnitten

Auf dem ins Anwendungsprogramm übertragenen Sequenzausschnitt kann mit Hilfe so genannter *Sequenzausschnittsmethoden* lesend und ändernd gearbeitet werden [Ott07]. Für unsere folgenden Ausführungen setzen wir Sequenzausschnittsmethoden voraus, die in ihrer Funktionsweise an die im Abschnitt 2.8 vorgestellte XML-Programmierschnittstelle DOM angelehnt sind (vgl. Abschnitt 6.5.1.1). Wie wir im Abschnitt 6.5.1 erläutern werden, sind prinzipiell jedoch auch andere Verarbeitungsansätze denkbar.

Wir nehmen hier an, dass es sich beim zweitlinkesten Baum des Sequenzausschnitts um das in Abbildung 3.30 dargestellte Bonusangebot handelt. Wir unterstellen ferner, dass sich der Kunde dafür entschieden hat, den einlösbaren Gegenwert dieses Bonusangebots um 50 Cent zu erhöhen.[6] Eine entsprechende Erhöhung des im selben Geschäft einlösbaren Gegenwerts ist dann wie in *Abbildung 4.18* gezeigt möglich. Der andere Gegenwert (also der Gegenwert, der bei einem Folgeeinkauf in einem anderen Geschäft gelten würde) lässt sich analog anpassen.

```
x = pSPM->sps(1)->tree(2)->e_child("GegenWert")->v_child(1)->getVal();

pSPM->sps(1)->tree(2)->e_child("GegenWert")->v_child(1)->setVal(x+0.50);
```

Abbildung 4.18: Lokales Erhöhen des einlösbaren Gegenwerts

In den in Abbildung 4.18 aufgeführten C++-Anweisungen dient die Variable *pSPM* als Zeiger auf das Objekt, welches für die lokale Verwaltung unseres Sequenzausschnitts zuständig ist (vgl. Abschnitt 4.2.6). Der Zugriff auf den ersten (und zugleich einzigen) Sequenzausschnittsteil erfolgt mit Hilfe des Methodenaufrufs *sps(1)*.[7] Mittels *tree(2)* wird der zweitlinkeste Baum dieses Sequenzausschnittsteils angesprochen. Ausgehend vom Wurzelknoten dieses Baumes gelangt man mit dem Methodenaufruf *e_child("GegenWert")* zum (linkesten) E-Kindknoten namens *GegenWert*. Auf dessen linkesten V-Kindknoten wird dann mittels *v_child(1)* zugegriffen. Die Aufrufe der Methoden *getVal* und *setVal* dienen zum Auslesen bzw. Setzen des entsprechenden (Gegen-)Werts.

Die vom Anwendungsprogramm am Sequenzausschnitt vorgenommenen Änderungen werden lokal (automatisch) protokolliert, um bei Bedarf später in die Datenbank eingebracht werden zu können. Das lokale Arbeiten auf dem Sequenzausschnitt mittels Methodenaufrufen erfolgt also zunächst ohne direkte Interaktion mit dem SQL-Laufzeitsystem, woraus sich eine Entlastung des SQL-Laufzeitsystems und damit eine bessere Skalierbarkeit des Gesamtsystems ergibt. Abschnitt 6.5.3 wird die Thematik der lokalen Änderungsprotokollierung weiter vertiefen.

4.2.8 Einbringen oder Verwerfen der lokalen Änderungen

Abschnitt 4.2.8.1 widmet sich zunächst dem Fall, dass die lokalen Änderungen in die Datenbank (wieder)eingebracht werden sollen. Die Alternative hierzu — nämlich das Verwerfen der lokalen Änderungen — wird anschließend im Abschnitt 4.2.8.2 betrachtet.

[6] In unserem Beispielszenario kann der Kunde bei jedem aufwertbaren Bonusangebot jeweils zwischen der Erhöhung des einlösbaren Gegenwerts um 50 Cent und einer Verlängerung der Gültigkeit um einen Monat wählen (Abschnitt 2.13).

[7] Das Kürzel *sps* steht für *sequence part section*, also *Sequenzausschnittsteil*.

4.2.8.1 Einbringen der lokalen Änderungen

Zunächst wird durch den in *Abbildung 4.19* dargestellten Methodenaufruf veranlasst, dass sämtliche den Sequenzausschnitt betreffenden Änderungsinformationen in einem zusammenhängenden (lokalen) Speicherbereich abgelegt werden (vgl. Abschnitt 6.5.3). Die in der abgebildeten C++-Anweisung genutzte Variable *pSPM* dient (wieder) als Zeiger auf das für die lokale Sequenzausschnittsverwaltung verantwortliche Objekt.

```
pSPM->provide_change_information();
```

Abbildung 4.19: Bereitstellen der lokalen Änderungsinformationen

Wie bei der (im Abschnitt 4.2.6 beschriebenen) Übertragung der Sequenzausschnitte wird die SQLSPCA auch hier zur Kommunikation zwischen Anwendungsprogramm und SQL-Laufzeitsystem genutzt: Unter Verwendung der SQLSPCA teilt das Anwendungsprogramm dem SQL-Laufzeitsystem sowohl die Adresse als auch die Größe des lokalen Speicherbereichs mit, der die lokalen Änderungsinformationen enthält *(Abbildung 4.20)*. Hierzu vermerkt das Anwendungsprogramm die Adresse des Speicherbereichs in der SQLSPCA-Komponente *pointer_to_local_storage* und die Größenangabe in der Komponente *size_of_required_storage*. Wir unterstellen hierbei, dass sich beide Angaben mit Hilfe der Methoden *get_pointer_to_change_information()* und *get_size_of_change_information()* ermitteln lassen.

```
sqlspca.pointer_to_local_storage =
  pSPM->get_pointer_to_change_information();

sqlspca.size_of_required_storage =
  pSPM->get_size_of_change_information();
```

Abbildung 4.20: Ablegen von Hilfsinformationen in der SQLSPCA

Das (eigentliche) Einbringen der lokalen Änderungen (auf das wir im Abschnitt 6.6 ausführlich eingehen werden) wird durch die in *Abbildung 4.21* gezeigte Anweisung veranlasst. Die Bezugnahme auf den entsprechenden Sequenzausschnitt (also den Sequenzausschnitt, in den die Änderungen eingebracht werden sollen) erfolgt dabei wie gewohnt über die Kombination aus Ergebnisspaltenname und Tupelcursorname. Nähere Informationen über die hier verwendete Sequenzcursoranweisung finden sich im Abschnitt 7.3.5.

```
BRING IN LOCAL CHANGES OF SEQUENCE PART
CONTAINED IN RESULT COLUMN Bonus
OF unserTC;
```

Abbildung 4.21: Einbringen der lokalen Änderungen

Beim Ausführen der Anweisung `BRING IN LOCAL CHANGES` greift das SQL-Laufzeitsystem auf die SQLSPCA zu, um sowohl die Adresse als auch die Größe des lokalen Speicherbereichs zu ermitteln, in dem die Änderungsinformationen enthalten sind. Anschließend liest das SQL-Laufzeitsystem die Änderungsinformationen aus diesem Speicherbereich aus.

Basierend auf den ausgelesenen Änderungsinformationen werden die lokalen Änderungen dann in die Datenbank eingebracht.

Nach dem Einbringen der lokalen Änderungen wird das zur lokalen Sequenzausschnittsverwaltung genutzte Objekt nicht länger benötigt und kann somit zerstört werden *(Abbildung 4.22)*.[8] Dabei werden (automatisch) auch die von diesem Objekt verwalteten Speicherbereiche wieder freigegeben.

```
delete pSPM;
```

Abbildung 4.22: Zerstören des nicht mehr benötigten Objekts

4.2.8.2 Verwerfen der lokalen Änderungen

Die lokal am Sequenzausschnitt vorgenommenen Änderungen können *explizit* oder *implizit* verworfen werden. Für ein *explizites* Verwerfen steht die in *Abbildung 4.23* dargestellte Anweisung zur Verfügung. Mit ihrer Hilfe wird dem SQL-Laufzeitsystem signalisiert, dass auf ein Einbringen der entsprechenden Änderungen „absichtlich“ verzichtet wird. Die Bezugnahme auf den betreffenden Sequenzausschnitt erfolgt dabei (wie üblich) über die Kombination aus Ergebnisspaltenname und Tupelcursorname. Für weitere Details bezüglich der hier genutzten Sequenzcursoranweisung verweisen wir auf Abschnitt 7.3.6.

```
DISCARD LOCAL CHANGES OF SEQUENCE PART
CONTAINED IN RESULT COLUMN Bonus
OF unserTC;
```

Abbildung 4.23: Verwerfen der lokalen Änderungen

Ein *implizites* Verwerfen der lokalen Änderungen ist möglich, indem der Tupelcursor weiterbewegt oder geschlossen wird, *ohne* dass die lokalen Änderungen zuvor in die Datenbank eingebracht wurden. Das implizite Verwerfen besitzt jedoch den Nachteil, dass nicht klar erkennbar ist, ob das Einbringen der lokalen Änderungen beabsichtigt oder unbeabsichtigt unterlassen wurde. Aus diesem Grund sollte für ein Verwerfen der lokalen Änderungen die explizite Variante bevorzugt werden.[9]

Nach einem Verwerfen der lokalen Änderungen wird das Objekt, welches zur lokalen Verwaltung des Sequenzausschnitts verwendet wurde, nicht mehr benötigt und kann zerstört werden. Abbildung 4.22 zeigt die hierfür zu nutzende C++-Anweisung.

4.2.9 Löschen der Sequenzcursor und Schließen des Tupelcursors

Nachdem das letzte Tupel unseres Anfrageergebnisses abgearbeitet wurde, werden die beiden Sequenzcursor *meinSC1* und *meinSC2* nicht länger gebraucht und können gelöscht

[8] Es ist prinzipiell denkbar, hier auf die Zerstörung des Objekts zu verzichten, um es für die Verwaltung eines anderen Sequenzausschnitts erneut benutzen zu können. Bei einem derartigen Vorgehen muss jedoch sichergestellt werden, dass der zur lokalen Aufnahme des Sequenzausschnitts vorgesehene (bei der Objekterzeugung allokierte) Speicherberich hinreichend groß dimensioniert ist.

[9] Es wäre denkbar, dass das SQL-Laufzeitsystem eine Warnung meldet, falls lokale Änderungen *implizit* verworfen werden.

werden *(Abbildung 4.24)*. Die hierzu vorgesehene Sequenzcursoranweisung `DROP SEQUENCE CURSORS` wird im Abschnitt 7.2.2 genauer vorgestellt.

```
DROP SEQUENCE CURSORS meinSC1, meinSC2;
```

Abbildung 4.24: Löschen der nicht mehr benötigten Sequenzcursor

Im Anschluss an das Löschen der beiden Sequenzcursor wird der Tupelcursor *unserTC* mittels der in *Abbildung 4.25* gezeigten (klassischen) SQL-Anweisung geschlossen.

```
CLOSE unserTC;
```

Abbildung 4.25: Schließen des Tupelcursors

4.3 Erweiterungen des Verarbeitungsablaufs

Wir werden im Folgenden kurz ein paar denkbare (und zugleich sinnvolle) Erweiterungen des in Abbildung 4.1 dargestellten Sequenzcursor-basierten Verarbeitungsablaufs erläutern. In *Abbildung 4.26* sind diese Erweiterungen in Form der gepunkteten Pfeile *E1* bis *E7* veranschaulicht.

- *Erweiterung E1*

  Ein Tupelcursor kann, wie aus SQL bekannt, nach seinem Schließen erneut geöffnet werden. Wie in Abschnitt 2.4 geschildert, ist es dabei möglich, die in der entsprechenden `SELECT`-Anfrage enthaltenen Hostvariablen mit anderen Werten zu belegen bzw. eine (völlig) andere `SELECT`-Anfrage an den Tupelcursor zu binden.

  Bezogen auf unser Beispielszenario hat ein derartiges Vorgehen den Vorteil, dass mit Hilfe ein und desselben Tupelcursors (nacheinander) die aufwertbaren Bonusangebote *mehrerer* Kunden verarbeitet werden können.

- *Erweiterung E2*

  Soll, nachdem die lokalen Änderungen sämtlicher im aktuellen Ergebnistupel enthaltenen Sequenzausschnitte eingebracht bzw. verworfen wurden, kein weiteres Ergebnistupel mehr verarbeitet werden, so können die Sequenzcursor gelöscht und der Tupelcursor geschlossen werden, *ohne* dass der Tupelcursor zuvor nochmals weiterbewegt wird. Ein nochmaliges Weiterbewegen des Tupelcursors hätte hier keinerlei Nutzen.

- *Erweiterung E3*

  Der auf ein Ergebnistupel positionierte Tupelcursor kann weiterbewegt werden, *ohne* dass die im entsprechenden Ergebnistupel enthaltenen XML-Werte zuvor (Sequenzcursor-basiert) verarbeitet werden [Böh07]. Beispielsweise kann nach Auswertung der Nicht-XML-Spaltenwerte des aktuellen Ergebnistupels entschieden werden, dass keine Verarbeitung der XML-Spaltenwerte stattfinden soll.

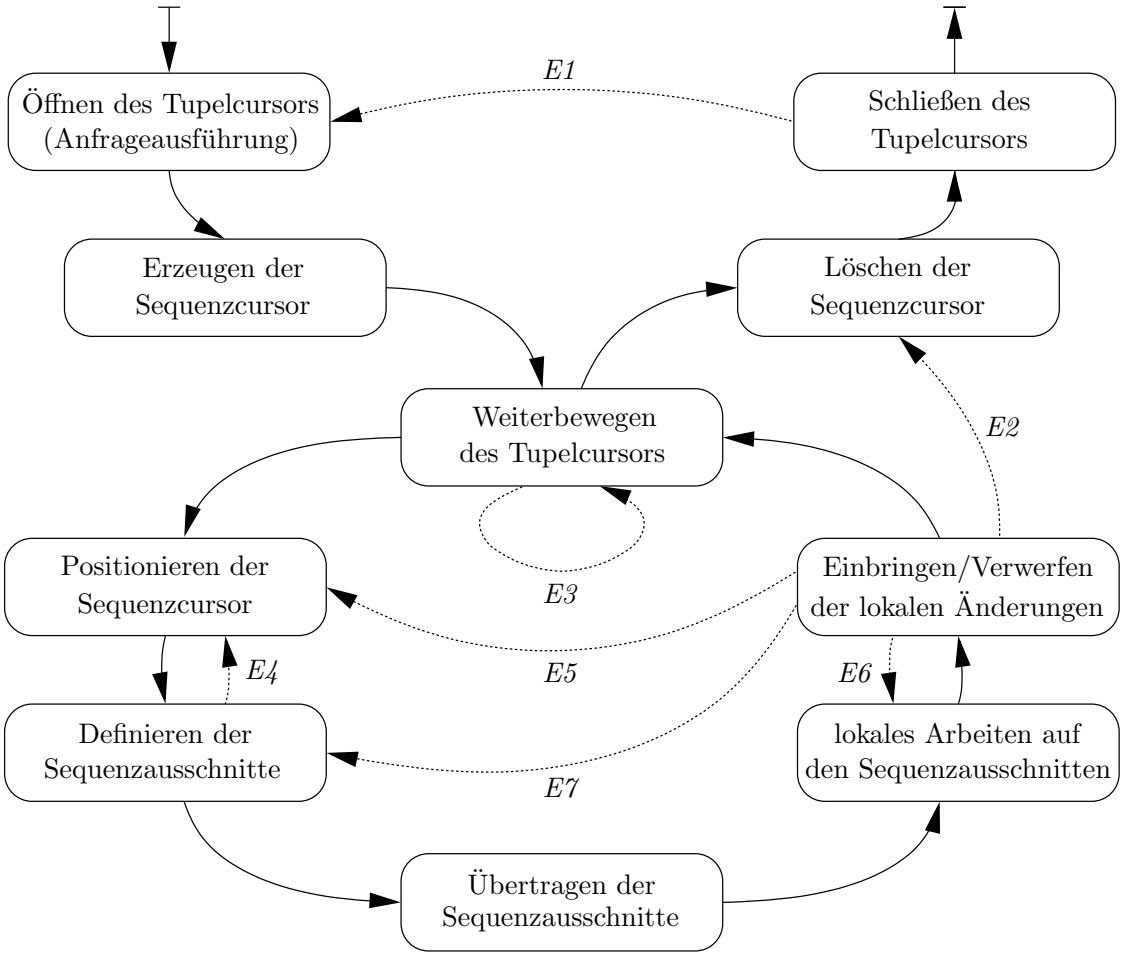


Abbildung 4.26: Erweiterter Sequenzcursor-basierter Verarbeitungsablauf

- *Erweiterung E4*

  Nach der Definition eines Sequenzausschnitts können die Sequenzcursor neu positioniert werden. Damit wird die Grundlage für eine sich anschließende (auf den neuen Sequenzcursorpositionen basierende) *Re*definition des entsprechenden Sequenzausschnitts geschaffen. Eine solche Redefinition kann sinnvoll sein, wenn z. B. nach Ausführung der Sequenzcursoranweisung `DESCRIBE SEQUENCE PART` festgestellt wird, dass zur lokalen Aufnahme des Sequenzausschnitts zu viel Speicherplatz benötigt würde.

- *Erweiterung E5*

  Nachdem die an einem Sequenzausschnitt lokal vorgenommenen Änderungen eingebracht oder verworfen wurden, können die Sequenzcursor neu positioniert werden. Mit Bezugnahme auf die neuen Sequenzcursorpositionen lässt sich dann eine Redefinition des entsprechenden Sequenzausschnitts durchführen.

  Eine Sequenzausschnitts-Redefinition ist beispielsweise notwendig, falls während des lokalen Arbeitens bemerkt wird, dass der ursprüngliche Sequenzausschnitt nicht alle lokal benötigten Knoten umfasst [Böh07].

- *Erweiterung E6*

  Nach dem Einbringen der lokalen Änderungen eines Sequenzausschnitts kann das lokale Arbeiten auf dem entsprechenden Sequenzausschnitt fortgesetzt werden. Damit ist es möglich, einen Verarbeitungs-Zwischenstand in der Datenbank zu sichern, der dann als Ausgangspunkt für das weitere lokale Arbeiten dient.

- *Erweiterung E7*

  Wurden die lokal an einem Sequenzausschnitt vorgenommenen Änderungen eingebracht oder verworfen, so kann sich daran unmittelbar eine Redefinition des entsprechenden Sequenzausschnitts anschließen. Im Gegensatz zur *Erweiterung E5* basiert diese Redefinition jedoch auf den *unveränderten* Sequenzcursorpositionen.

  Eine derartige Redefinition ist natürlich nur dann sinnvoll, wenn sich der neu definierte Sequenzausschnitt vom ursprünglichen Sequenzausschnitt unterscheidet. Dass dies trotz der unveränderten Sequenzcursorpositionen tatsächlich möglich ist, soll folgendes Beispiel zeigen:

  Wir nehmen an, dass das aktuelle Ergebnistupel eine aus sieben Bäumen bestehende Sequenz enthält, wobei ein Sequenzcursor auf die Wurzel des letzten Baums und ein anderer Sequenzcursor auf die Wurzel des fünftletzten Baums positioniert ist (vgl. Abbildung 4.11). Wir unterstellen ferner, dass mit Bezugnahme auf die beiden Sequenzcursor zunächst ein (nicht zusammenhängender) Sequenzausschnitt definiert wurde, der nur den fünftletzten sowie den letzten Baum umfasst. Wird nun festgestellt, dass für die lokale Verarbeitung auch die dazwischenliegenden drei Bäume benötigt werden, so lässt sich basierend auf den unveränderten Sequenzcursorpositionen ein (zusammenhängender) Sequenzausschnitt definieren, zu dem *jeder* der letzten fünf Bäume gehört (vgl. Abbildung 4.13).

4.4 Denkbare alternative Verarbeitungsansätze

Im aktuellen Abschnitt fassen wir die wesentlichen Eigenschaften der Sequenzcursor-basierten Verarbeitung nochmals kompakt zusammen. Anhand der einzelnen Eigenschaften diskutieren wir dabei denkbare Alternativen zum Sequenzcursor-basierten Verarbeitungsmodell.

- *Lokale Verarbeitung im Anwendungsprogramm*

  Bei dem von uns vorgestellten Sequenzcursor-basierten Verfahren werden Ausschnitte von (XQuery-)Sequenzen ins Anwendungsprogramm übertragen, um dort *lokal* verarbeitet zu werden. Die Alternative hierzu wäre, auf eine derartige Übertragung zu verzichten und eine *datenbankseitige* Verarbeitung vorzunehmen.

  Für gewisse Anforderungen ist diese Alternative durchaus geeignet. Soll beispielsweise ermittelt werden, über wie viele Bonusangebote ein bestimmter Kunde in einer gewissen Kategorie verfügt, so ist es dazu nicht erforderlich, die entsprechende Bonusangebotesequenz ins Anwendungsprogramm zu übertragen. Stattdessen könnte mit Hilfe der SQL-Funktion `XMLQUERY` *datenbankseitig* ermittelt werden, aus wie vielen Bäumen die entsprechende Sequenz besteht. Es würde dann genügen, das Ergebnis dieser datenbankseitigen Verarbeitung (bei welchem es sich um einen einzelnen Integer-Wert handelt) ins Anwendungsprogramm zu übertragen.

Es existieren jedoch Szenarien, in denen eine datenbankseitige Verarbeitung ungeeignet ist. Hierzu zählt z. B. die vom Kunden vorzunehmende Aufwertung seiner Bonusangebote. Durch die angebotene Wahlmöglichkeit (Erhöhung des Gegenwerts oder Verlängerung der Gültigkeit) muss der Kunde für jedes seiner aufwertbaren Bonusangebote jeweils separat entscheiden, welche Art der Aufwertung er nutzen will. Als Grundlage für seine Entscheidungen sollten dem Kunden dabei sämtliche Details der aufwertbaren Bonusangebote (graphisch aufbereitet) am Bildschirm angezeigt werden. Um dies zu realisieren, werden die letzten fünf Bäume der entsprechenden Bonusangebotesequenz im (für die graphische Aufbereitung zuständigen) Anwendungsprogramm benötigt. Es muss hier also eine Übertragung ins Anwendungsprogramm stattfinden, an die sich (zunächst) eine lesende lokale Verarbeitung anschließt.

Gemäß des Sequenzcursor-basierten Verarbeitungsmodells kann auf dem entsprechenden (ins Anwendungsprogramm übertragenen) Sequenzausschnitt aber nicht nur lesend, sondern auch ändernd gearbeitet werden. Damit ist es für das Anwendungsprogramm sehr leicht möglich, die aus der Aufwertung der Bonusangebote resultierenden Änderungen (lokal) in den Sequenzausschnitt einzuarbeiten. Hierzu stehen dem Anwendungsprogramm komfortable Sequenzausschnittsmethoden zur Verfügung. Mit Hilfe der Sequenzcursoranweisung `BRING IN LOCAL CHANGES` kann das Anwendungsprogramm anschließend veranlassen, dass diese Änderungen in die Datenbank eingebracht werden.

Alternativ zum zuvor beschriebenen Vorgehen könnte das Anwendungsprogramm die notwendigen Änderungen auch direkt *datenbankseitig* vornehmen — also ohne den „Umweg“ über eine lokale Verarbeitung. Hierzu würde das Anwendungsprogramm eine `UPDATE`-Anweisung nutzen, mit deren Hilfe die entsprechende (in der Basistabelle *Kukabo* enthaltene) Bonusangebotesequenz aktualisiert wird. Die Schwierigkeit bestände dabei aber darin, einen geeigneten Ausdruck für die rechte Seite der (zur `UPDATE`-Anweisung gehörenden) `SET`-Klausel bereitzustellen. Der Wert dieses Ausdrucks müsste der aktualisierten Bonusangebotesequenz (also der Bonusangebotesequenz *nach* Durchführung der Aufwertungen) entsprechen. Da die Generierung eines solchen Ausdrucks sehr aufwendig und fehleranfällig wäre, ist die Verwendung der `UPDATE`-Anweisung keine akzeptable Alternative zur (im Rahmen der Sequenzcursor-basierten Verarbeitung vorgesehenen) lokalen Verarbeitung.

- *Vollständige Sequenzen im Anfrageergebnis*

  Das bei der Sequenzcursor-basierten Verarbeitung zu Beginn erzeugte Anfrageergebnis enthält i. d. R. die *vollständigen* (Ursprungs-)Sequenzen — also nicht nur Ausschnitte der in der zugrunde liegenden Basistabelle vorkommenden Sequenzen. Die Festlegung der ins Anwendungsprogramm zu übertragenden Sequenz*ausschnitte* erfolgt (mit Hilfe von Sequenzcursorn) erst zu einem späteren Zeitpunkt.

  Alternativ wäre es denkbar, die `SELECT`-Anfrage so zu formulieren, dass von vornherein nur die für die weitere Verarbeitung relevanten Sequenzausschnitte im Anfrageergebnis enthalten sind. Hierzu müsste innerhalb der `SELECT`-Anfrage die SQL-Funktion `XMLQUERY` genutzt werden, mit deren Hilfe sich die gewünschten Sequenzausschnitte spezifizieren ließen. Die Festlegung der Sequenzausschnitte würde dabei mit Hilfe von XQuery-Anfragen realisiert. Dies hätte jedoch die Konsequenz, dass (lokal) an den Sequenzausschnitten vorgenommene Änderungen im Allgemeinen nicht

auf die Ursprungs-Sequenzen rückabbildbar wären. Damit könnten die entsprechenden Änderungen nicht in die Datenbank eingebracht werden.

Sofern im Anwendungsprogramm also *ändernd* auf den Sequenzausschnitten gearbeitet werden soll, müssen die entsprechenden Sequenzen jeweils vollständig im Anfrageergebnis enthalten sein. Falls die Sequenzausschnitte hingegen nur *rein lesend* verarbeitet werden, ist es prinzipiell möglich, bereits mit Hilfe der `SELECT`-Anfrage eine Einschränkung auf Sequenzausschnitte vorzunehmen.

- *Keine Übertragung der vollständigen Sequenzen*

  Gemäß des Sequenzcursor-basierten Verarbeitungsmodells wird eine im Anfrageergebnis enthaltene Sequenz i. d. R. nicht komplett ins Anwendungsprogramm übertragen — stattdessen erfolgt nur die Übertragung eines Sequenz*ausschnitts*. Die Alternative hierzu wäre, die Sequenz komplett zu übertragen.

  SQL-Anfrageergebnisse können u. U. sehr große Sequenzen enthalten. In vielen denkbaren Anwendungsszenarien (wie beispielsweise in unserem Kundenkartenszenario) sind für die weitere Verarbeitung aber nicht die vollständigen Sequenzen, sondern nur Ausschnitte dieser Sequenzen von Interesse [Mül05]. Im vorgestellten Beispielszenario werden für die lokale Verarbeitung beispielsweise nur die jeweils letzten fünf Bäume der Sequenzen benötigt.

  Sofern es genügt, lokal lediglich Ausschnitte der im Anfrageergebnis enthaltenen Sequenzen vorzuhalten, ist es aus Performance- und Speicherplatzerwägungen sinnvoll, nur diese Sequenzausschnitte (anstelle der vollständigen Sequenzen) ins Anwendungsprogramm zu übertragen. Insbesondere die damit erzielbare Begrenzung des Übertragungsaufwands spricht für ein derartiges Vorgehen (und somit gegen die Alternative, eine komplette Übertragung vorzunehmen).

- *Explizite Festlegung von zu übertragenden Sequenzausschnitten*

  Der Sequenzcursor-basierte Verarbeitungsablauf sieht eine explizite Festlegung der ins Anwendungsprogramm zu übertragenden Sequenzausschnitte vor. Das eigentliche lokale Arbeiten erfolgt dann auf den entsprechend definierten Sequenzausschnitten.

  Die sich hier anbietende Alternative bestände darin, auf die explizite Festlegung von Sequenzausschnitten zu verzichten. Das lokale Arbeiten würde damit (zumindest aus logischer Sicht) auf den *Gesamt*sequenzen erfolgen. Ein derartiges Vorgehen hätte den Vorteil, dass das Anwendungsprogramm nicht für die Definition von Sequenzausschnitten verantwortlich wäre.

  Auch bei einem Verzicht auf die explizite Festlegung von Sequenzausschnitten müsste sichergestellt werden, dass gewisse (für die lokale Verarbeitung benötigte) Sequenzteile[10] lokal verfügbar sind. Dazu sollte es nicht erforderlich sein, die Sequenzen komplett ins Anwendungsprogramm zu übertragen. Es müsste also eine *automatische* (und für das Anwendungsprogramm transparente) Festlegung und Verwaltung von lokal vorzuhaltenden Sequenzteilen durchgeführt werden.

  Bei einer solchen automatischen Festlegung von Sequenzteilen könnte aber das (oftmals vorhandene) Anwendungswissen über die *tatsächlich* benötigten Sequenzteile

[10] Wir verwenden hier bewusst den Begriff „Sequenz*teile*“ und nicht „Sequenz*ausschnitte*“. Unter einem Sequenz*ausschnitt* verstehen wir im Rahmen dieser Arbeit einen *explizit* vom Anwendungsprogramm festgelegten Ausschnitt einer Sequenz. Sequenz*teile* werden hingegen nicht explizit vom Anwendungsprogramm festgelegt.

nicht genutzt werden. Dies hätte mit hoher Wahrscheinlichkeit zur Folge, dass die lokal vorgehaltenen (automatisch ermittelten) Sequenzteile einerseits Knoten enthalten, die lokal gar nicht gebraucht werden, während andererseits einige lokal benötigte Knoten zunächst fehlen würden und später nachgeholt werden müssten. Damit wären im Wesentlichen zwei Nachteile verbunden: ein erhöhter lokaler Speicherbedarf und ein größerer Übertragungsaufwand.

Falls (wie in unserem Beispielszenario) feststeht, welche Teile der im Anfrageergebnis enthaltenen Sequenzen zur lokalen Verarbeitung benötigt werden, besitzt das automatische Ermitteln lokal vorzuhaltender Sequenzteile also deutliche Nachteile gegenüber dem (vom Sequenzcursor-basierten Verarbeitungsablauf vorgesehenen) expliziten Festlegen von Sequenzausschnitten. In Anwendungsszenarien, in denen es hingegen nicht (oder nur schwer) möglich ist, eine Aussage über die lokal benötigten Sequenzteile zu treffen, sollte anstelle der Sequenzcursor-basierten Verarbeitung eine alternative Vorgehensweise genutzt werden, bei der die lokal vorgehaltenen Sequenzteile automatisch bestimmt werden.

- *Festlegung der Sequenzausschnitte mit Hilfe von Sequenzcursorn*

  Im Rahmen der Sequenzcursor-basierten Verarbeitung erfolgt die Festlegung der ins Anwendungsprogramm zu übertragenden Sequenzausschnitte mit Hilfe von Sequenzcursorn, die hierfür zuvor innerhalb der Sequenzen positioniert werden. Alternativ wäre es denkbar, die Festlegung der Sequenzausschnitte nicht cursorbasiert, sondern mit Hilfe geeigneter deskriptiver Anfragen vorzunehmen.

  Gegen diese Alternative spricht jedoch, dass die Auswahl der Sequenzausschnitte stets basierend auf dem *Ergebnis* einer `SELECT`-Anfrage erfolgt. Während des Abarbeitens dieses Anfrageergebnisses müssten dann Anfragen gegen einzelne (im Anfrageergebnis enthaltene) Spaltenwerte gestellt und ausgewertet werden. Die Verarbeitung des (ursprünglichen) Anfrageergebnisses würde sich dadurch deutlich verkomplizieren. Außerdem würde ein derartiges Vorgehen den allgemeinen SQL-Sprachgewohnheiten widersprechen: SQL sieht für das Abarbeiten eines Anfrageergebnisses zwar den Einsatz von Cursorn, nicht jedoch das Auswerten von (auf dem Anfrageergebnis aufsetzenden) Anfragen vor.

- *Auf typed-value-orientierter Repräsentation beruhender Ablauf*

  Die Sequenzcursor-basierte Verarbeitung beruht vollständig auf der von uns konzipierten typed-value-orientierten Sequenzrepräsentation. Die Alternative hierzu — nämlich eine Nutzung der traditionellen Sequenzrepräsentation als Grundlage für die Sequenzcursor-basierte Verarbeitung — wäre mit erheblichen Nachteilen verbunden. Für nähere Details diesbezüglich verweisen wir auf Abschnitt 3.1.1.

4.5 Abgrenzung gegen existierende Ansätze

Der am Anfang des aktuellen Kapitels vorgestellte Sequenzcursor-basierte Verarbeitungsablauf erlaubt es, (Ausschnitte von) XQuery-Sequenzen, die als Spaltenwerte eines SQL-Anfrageergebnisses auftreten, in einem Anwendungsprogramm zu verarbeiten. Wir werden dieses von uns entwickelte Verfahren im Folgenden gegen verschiedene existierende Ansätze abgrenzen, die ebenfalls eine Verarbeitung von (in SQL-Anfrageergebnissen enthaltenen) XML-Werten ermöglichen oder zumindest gewisse Bezüge dazu aufweisen.

4.5.1 Serialisierte Übergabe von XML-Werten

SQL/XML:2006 bietet die Möglichkeit, die im Anfrageergebnis enthaltenen XML-Werte als „normale“ Zeichenketten ins Anwendungsprogramm zu übertragen. Hierzu werden die XML-Werte bei der Übertragung implizit mittels der SQL-Funktion `XMLSERIALIZE` [ZRE+03, MB06] in Zeichenketten serialisiert und dann als `VARCHAR`- oder `CLOB`-Werte ans Anwendungsprogramm übergeben [Kul03, ISO06].

Im Gegensatz zur Sequenzcursor-basierten Verarbeitung ist das von der SQL-Norm vorgesehene Übergabeverfahren aber nicht für beliebige XML-Werte nutzbar: SQL/XML:2006 beschränkt die serialisierte Übergabe auf Sequenzen, die aus (maximal) einem Sequenzeintrag bestehen. Sequenzen mit *mehreren* Sequenzeinträgen lassen sich also nicht serialisiert ins Anwendungsprogramm übertragen. Für unser Beispielszenario (bei dem die im Anfrageergebnis enthaltenen Sequenzen i. d. R. mehrere Sequenzeinträge umfassen) ist das Verfahren damit ungeeignet.

Die SQL-Normungsgremien sind sich dieser mit ihrem Verfahren verbundenen Einschränkung übrigens durchaus bewusst und planen für künftige SQL-Versionen eine entsprechende Erweiterung [Zem04a]. Somit ist zu erwarten, dass sich in folgenden SQL-Versionen *sämtliche* XML-Werte serialisiert ins Anwendungsprogramm übertragen lassen.

Gegenüber der von uns vorgeschlagenen Sequenzcursor-basierten Verarbeitung besitzt die serialisierte Übergabe von XML-Werten aber noch einen weiteren entscheidenden Nachteil: Bei der Übertragung ins Anwendungsprogramm gehen alle Typinformationen (mit denen die Knoten der Sequenz angereichert sind) verloren. Im Anwendungsprogramm ist eine adäquate Verarbeitung der übertragenen XML-Werte somit nicht unmittelbar möglich. Um die (für eine lokale Verarbeitung notwendigen) Typinformationen nutzen zu können, müsste das Anwendungsprogramm die als Zeichenketten vorliegenden XML-Werte zunächst neu parsen und anschließend einer Validierung (einschließlich der damit verbundenen Anreicherung mit Typinformationen) unterziehen. Dies wäre jedoch mit einem enormen Aufwand sowie deutlichen Performanceeinbußen verbunden.

Da die Typinformationen ein integraler Bestandteil einer XQuery-Sequenz sind, muss der Verlust der Typinformationen (unabhängig von den dadurch hervorgerufenen Performanceeinbußen) als wesentliches Manko des von der SQL-Norm vorgesehenen Übergabeverfahrens angesehen werden. Die serialisierte Übergabe von XML-Werten hat (in gewissen Anwendungsszenarien) aber dennoch ihre Berechtigung: Sofern für die lokale Verarbeitung keinerlei Typinformationen gebraucht werden, können (und sollten) die XML-Werte serialisiert übergeben werden.

4.5.2 Nutzung der SQL-Funktion `XMLTABLE`

Zum Zugriff auf XML-Werte steht in SQL/XML:2006 (u. a.) die Funktion `XMLTABLE` zur Verfügung [ZRK+04, MB06]. Mit Hilfe dieser (in der `FROM`-Klausel einer SQL-`SELECT`-Anfrage zu nutzenden) Tabellenfunktion ist es möglich, auf einen XML-Wert zuzugreifen, als würde es sich bei diesem um eine (Ergebnis-)Tabelle handeln. Hierzu wird beim Aufruf der `XMLTABLE`-Funktion eine (aus einer XQuery-Anfrage und mehreren XPath-Ausdrücken bestehende) Transformationsvorschrift angegeben, welche bestimmte Teile des XML-Werts in Spaltenwerte der entsprechenden Ergebnistabelle überführt.

Im Gegensatz zum Sequenzcursor-basierten Verarbeitungsmodell erfolgt die weitere Verarbeitung einer XQuery-Sequenz hier also nicht in ihrer „natürlichen Form als Sequenz“, sondern als Tabelle. Dabei ist es fraglich, ob sich stets passende Transformationsvorschriften finden lassen, mit denen die Sequenzen in geeignete Tabellen überführt werden können. Selbst wenn derartige Transformationsvorschriften *theoretisch* existieren, ist ihre *tatsächliche* Ermittlung eine u. U. nicht-triviale Aufgabe.

Gegenüber der Sequenzcursor-basierten Verarbeitung besitzt die Nutzung der `XMLTABLE`-Funktion insbesondere die folgenden zwei Nachteile: Zum einen wird die `SELECT`-Anfrage deutlich verkompliziert, da ja der Aufruf der `XMLTABLE`-Funktion (bei dem eine XQuery-Anfrage und mehrere XPath-Ausdrücke als Parameter übergeben werden) *innerhalb* der `SELECT`-Anfrage erfolgt. Zum anderen (und das ist der weit schwerwiegendere Nachteil) kann auf den mittels `XMLTABLE` erzeugten Ergebnistabellen nur lesend gearbeitet werden. Es ist also nicht möglich, Änderungen an den Spaltenwerten der Ergebnistabellen vorzunehmen, die dann auf die zugrunde liegenden Sequenzen (rück)abgebildet werden. Bezogen auf unser Beispielszenario wäre eine Aufwertung der in der *Kukabo*-Tabelle enthaltenen Bonusangebote also nicht realisierbar. Damit ist dieses Verfahren für unser Beispielszenario ungeeignet.

4.5.3 DOM, SAX und StAX

DOM (Document Object Model), *SAX (Simple API for XML)* und *StAX (Streaming API for XML)* sind Programmierschnittstellen (APIs) für den Zugriff auf XML-Dokumente. DOM (Abschnitt 2.8) arbeitet Baum-basiert: Das zu verarbeitende XML-Dokument wird als Baum aufgefasst, durch den das Anwendungsprogramm navigieren kann. Beim Navigieren kann das Anwendungsprogramm dann auf die im XML-Dokument enthaltenen Informationen zugreifen. SAX [MB02, KM02] verfolgt hingegen einen Ereignis-orientierten Ansatz. Das XML-Dokument wird vom SAX-Parser sequentiell gelesen, wobei bestimmte Ereignisse (z. B. der Beginn oder das Ende eines Elements) erkannt werden. Das Anwendungsprogramm kann nun auf diese Ereignisse reagieren. Auch StAX [BEA03, MB06] arbeitet Ereignis-orientiert. Im Gegensatz zu SAX reagiert das Anwendungsprogramm dabei aber nicht auf (automatisch vom Parser erkannte) Ereignisse, sondern kann explizit das jeweils nächste Ereignis (vom Parser) anfordern.

Wie das Sequenzcursor-basierte Verarbeitungsmodell ermöglichen auch DOM, SAX und StAX eine Verarbeitung von XML-Werten in Anwendungsprogrammen. Dennoch bestehen zwischen dem von uns konzipierten Verfahren und den drei genannten Programmierschnittstellen wesentliche Unterschiede:

- Der Sequenzcursor-basierte Verarbeitungsablauf dient zur Verarbeitung von XML-Werten, die in SQL-Anfrageergebnissen enthalten sind. Es handelt sich also um einen *integrativen* Ansatz, der die Verarbeitung von XML-Werten *im Kontext einer SQL-basierten Verarbeitung* gestattet. DOM, SAX und StAX sind hingegen *reine* XML-Programmierschnittstellen ohne jeglichen Bezug zu SQL.
- Das von uns konzipierte Verfahren ermöglicht die Verarbeitung von beliebigen XQuery-Sequenzen. Somit lassen sich auch XQuery-Sequenzen verarbeiten, die aus mehreren Sequenzeinträgen bestehen bzw. deren Knoten mit Typinformationen angereichert sind. Im Gegensatz hierzu erlauben DOM, SAX und StAX lediglich eine Verarbeitung von XML-Dokumenten.

Trotz der bestehenden Unterschiede lassen sich die Konzepte der hier betrachteten XML-Programmierschnittstellen im Rahmen der Sequenzcursor-basierten Verarbeitung nutzen. Beispielsweise haben wir im Abschnitt 4.2.7 Sequenzausschnittsmethoden verwendet, die in ihrer Funktionsweise an DOM angelehnt sind.

4.5.4 XQJ

XQJ (XQuery API for Java) [Mel07, EM04b] ist eine Programmierschnittstelle (API), die es ermöglicht, aus einem JAVA-Programm heraus auf eine *XQuery-Implementierung* zuzugreifen. Eine XQuery-Implementierung ist (vereinfacht betrachtet) das Analogon zu einem relationalen DBMS: Während ein relationales DBMS SQL-Anfragen ausführt, ist ein XQuery-System für die Auswertung von XQuery-Anfragen verantwortlich.

Mit Hilfe von XQJ kann ein Anwendungsprogramm die Ausführung einer XQuery-Anfrage veranlassen. Anschließend lässt sich das Ergebnis dieser Anfrage (bei welchem es sich um eine XQuery-Sequenz handelt) im Anwendungsprogramm verarbeiten. Hierzu stellt XQJ einen Cursor zum eintragsweisen Durchlaufen der XQuery-(Ergebnis-)Sequenz bereit. Beim gewünschten Sequenzeintrag angekommen, kann die weitere Verarbeitung dann z. B. mittels DOM, SAX oder StAX (vgl. Abschnitt 4.5.3) erfolgen.

Wie das Sequenzcursor-basierte Verfahren ermöglicht XQJ also eine Verarbeitung von XQuery-Sequenzen im Anwendungsprogramm. Außerdem steht ein Cursor zur Verfügung, mit dem in die Sequenzen eingetaucht werden kann. Trotz dieser Gemeinsamkeiten gibt es zwischen XQJ und dem Sequenzcursor-basierten Verarbeitungsmodell wesentliche Unterschiede. XQJ unterscheidet sich insbesondere wie folgt von dem von uns konzipierten Verfahren:

- XQJ besitzt keinerlei SQL-Bezug. Es geht also nicht um die Verarbeitung von Sequenzen, die in einem SQL-Anfrageergebnis enthalten sind, sondern um das Arbeiten auf einzelnen Sequenzen (außerhalb des Kontextes einer SQL-basierten Verarbeitung).
- Der von XQJ bereitgestellte Cursor dient lediglich zur Kennzeichnung des aktuell zu verarbeitenden Sequenzeintrags. Er wird also nicht genutzt, um Ausschnitte der Sequenzen zu definieren. Damit unterscheidet sich dieser Cursor in seiner Funktion wesentlich von unseren Sequenzcursorn.
- In XQJ kann keine explizite Festlegung von (ins Anwendungsprogramm zu übertragenden) Sequenzausschnitten vorgenommen werden.
- XQJ basiert nicht auf der typed-value-orientierten Sequenzrepräsentation, sondern auf der traditionellen Repräsentation.

4.5.5 XJ

XJ (XML Enhancements to Java) [HRS+05] ist eine im Rahmen eines Forschungsprojekts entwickelte Java-ähnliche Programmiersprache, die es erlaubt, XML-Werte komfortabel im Anwendungsprogramm zu verarbeiten. Zum Arbeiten auf den XML-Werten stellt XJ verschiedene Sprachkonstrukte bereit. So ist es beispielsweise möglich, XPath-1.0-Ausdrücke

auszuwerten oder gezielt Änderungen an Teilen der XML-Werte vorzunehmen. Eine Besonderheit von XJ besteht dabei darin, dass es sich bei den zur XML-Verarbeitung angebotenen Konstrukten um Sprachkonstrukte *„1. Klasse“* handelt.

Wie das Sequenzcursor-basierte Verarbeitungsmodell verfolgt XJ das Ziel, eine Verarbeitung von XML-Werten im Anwendungsprogramm zu ermöglichen. XJ besitzt — im Gegensatz zu dem von uns entwickelten Verfahren — aber keinerlei Datenbank- oder SQL-Bezüge: Das Arbeiten auf den XML-Werten erfolgt bei XJ also nicht im Kontext einer SQL-basierten Verarbeitung.

Da es sich bei XJ um eine eigenständige Programmiersprache handelt, müssen XJ-Programme mit Hilfe eines speziellen XJ-Compilers übersetzt werden. Anders als bei den im Rahmen der Sequenzcursor-basierten Verarbeitung genutzten Anwendungsprogrammen ist hier also kein Rückgriff auf bewährte „Standardcompiler“ möglich.

Während das von uns konzipierte Verfahren auf dem XQuery-Datenmodell aufbaut und damit die Verarbeitung beliebiger XQuery-Sequenzen gestattet, beschränkt sich XJ auf XML-Dokumente. XQuery-Sequenzen, die *mehrere* Bäume umfassen (wie beispielsweise die Bonusangebotesequenzen unseres Beispielszenarios) lassen sich mit XJ also nicht verarbeiten.

4.5.6 $\text{XOBE}_{\text{DBPL}}$

Das Forschungsprojekt *XOBE_{DBPL} (XML OBjEcts DataBase Programming Language)* [KL03, SL05, Sch05] der Universität Lübeck verfolgt einen ähnlichen Ansatz wie XJ: Die Programmiersprache Java wurde um Konstrukte erweitert, die eine Verarbeitung von XML-Werten im Anwendungsprogramm ermöglichen. So kann zum einen mittels XPath-1.0-Ausdrücken auf den XML-Werten gearbeitet werden, zum anderen lassen sich Änderungen an den XML-Werten vornehmen. $\text{XOBE}_{\text{DBPL}}$ — welches (wie XJ) keinerlei SQL-Bezüge aufweist — unterstützt (wie XJ) lediglich XML-Dokumente (und somit insbesondere keine aus mehreren Einträgen bestehenden XQuery-Sequenzen).

$\text{XOBE}_{\text{DBPL}}$ und XJ sind sich (in den für unsere Betrachtungen relevanten Punkten) sehr ähnlich. Für eine Abgrenzung von $\text{XOBE}_{\text{DBPL}}$ gegen den Sequenzcursor-basierten Verarbeitungablauf verweisen wir deshalb auf die entsprechenden Ausführungen des Abschnitts 4.5.5.

Kapitel 5

Sequenzcursor

Bei den Sequenzcursorn [Rab05, MR05, Mül06] handelt es sich um eine von uns neu eingeführte Cursorart. Sequenzcursor werden im Rahmen der (ausführlich im Kapitel 4 vorgestellten) Sequenzcursor-basierten Verarbeitung genutzt, um die ins Anwendungsprogramm zu übertragenden Sequenzausschnitte festzulegen. Hierzu werden die Sequenzcursor zunächst innerhalb der Sequenzen geeignet positioniert, anschließend erfolgt dann die Definition der Sequenzausschnitte mit Bezugnahme auf die Sequenzcursor.

Im aktuellen Kapitel werden wir näher auf die Sequenzcursor eingehen. Nach einer Betrachtung ihrer grundlegenden Eigenschaften im Abschnitt 5.1 beschreiben wir im Abschnitt 5.2, welche Möglichkeiten es für ihre Positionierung gibt. Die Positionierung eines Sequenzcursors kann erfolgreich sein oder scheitern. Nähere Erläuterungen hierzu enthält Abschnitt 5.3. Im Anschluss widmet sich Abschnitt 5.4 der Frage, welche Möglichkeiten für den Status eines Sequenzcursors existieren.

5.1 Grundlegende Eigenschaften

In den sich hier anschließenden Abschnitten 5.1.1 bis 5.1.4 werden wir die grundlegenden Eigenschaften der Sequenzcursor näher erläutern. Dabei werden wir jeweils auch kurz die denkbaren Alternativen diskutieren.

5.1.1 Bindung an Spalte vom Anfrageergebnis

Ein Anwendungsprogramm kann gleichzeitig auf mehreren Anfrageergebnissen arbeiten, wobei jedes dieser Anfrageergebnisse eine oder mehrere XML-Spalten besitzen kann. Jedem Anfrageergebnis ist jeweils genau ein Tupelcursor zugeordnet.

Beim Erzeugen eines Sequenzcursors wird dieser an eine bestimmte XML-Spalte eines konkreten Anfrageergebnisses gebunden [Ott07]. Hierzu wird in der Sequenzcursoranweisung `CREATE SEQUENCE CURSOR` (Abschnitt 7.2.1) sowohl der Spaltenname als auch der Tupelcursorname angegeben. Die Angabe des Tupelcursornamens dient dabei zur Bezugnahme auf ein bestimmtes Anfrageergebnis. Anstelle des Spaltennamens kann alternativ (z. B. wenn mehrere gleichnamige Ergebnisspalten existieren) auch die Spaltennummer verwendet werden.

Gemäß des Sequenzcursor-basierten Verarbeitungsmodells können Sequenzcursor stets nur innerhalb des jeweils aktuellen Ergebnistupels positioniert werden. Aus diesem Grund werden Sequenzcursor bei ihrer Erzeugung *nicht* an ein bestimmtes Ergebnistupel gebunden. Das Ergebnistupel, in dem (zu einem bestimmten Zeitpunkt) eine Positionierung möglich ist, ergibt sich stattdessen implizit aus der (zu diesem Zeitpunkt) aktuellen Position des entsprechenden Tupelcursors. Ein Sequenzcursor wird bei seiner Erzeugung also explizit an eine Ergebnisspalte, nicht aber an eine Ergebniszeile (ein Ergebnistupel) gebunden.

Zur (bei der Erzeugung eines Sequenzcursors stattfindenden) Bindung an eine bestimmte Ergebnisspalte wären prinzipiell die folgenden (im Rahmen der Sequenzcursor-basierten Verarbeitung *nicht* genutzten) Alternativen denkbar:

- *Alternative 1: Bindung an ein gesamtes Anfrageergebnis*

  Beim Erzeugen eines Sequenzcursors wird dieser zwar an ein bestimmtes Anfrageergebnis, nicht jedoch an eine konkrete Spalte dieses Anfrageergebnisses gebunden.

  Ein und derselbe Sequenzcursor kann damit (nacheinander) für verschiedene XML-Spalten des Anfrageergebnisses genutzt werden. Diese Flexibilität ist jedoch nur ein scheinbarer Vorteil: Beim Vorhandensein mehrerer XML-Ergebnisspalten werden i. d. R. in sämtlichen dieser Spalten *zugleich* Sequenzcursor benötigt. Da ein und derselbe Sequenzcursor aber nicht gleichzeitig in mehreren XML-Ergebnisspalten positioniert sein kann, sind für jede der entsprechenden Ergebnisspalten ohnehin separate Sequenzcursor erforderlich.

  Wird ein (nicht an eine Ergebnisspalte gebundener) Sequenzcursor positioniert, so muss dabei (u. a.) eine bestimmte Ergebnisspalte spezifiziert werden. Im Vergleich zum von uns vorgeschlagenen Vorgehen besitzen die Positionierungsanweisungen damit eine erhöhte Komplexität, was zum einen einen höheren Ausführungsaufwand und zum anderen eine höhere Anfälligkeit für Laufzeitfehler nach sich zieht: Bei der Positionierung eines Sequenzcursors muss nicht nur geprüft werden, ob der anzuspringende Knoten existiert, sondern es muss u. a. auch überprüft werden, ob das Anfrageergebnis überhaupt eine entsprechende Spalte besitzt und ob es sich dabei um eine XML-Spalte handelt.

- *Alternative 2: Keinerlei Bindung*

  Ein Sequenzcursor wird beim Erzeugen nicht an ein bestimmtes Anfrageergebnis (und somit erst recht nicht an eine konkrete Ergebnisspalte) gebunden.

  Dieser Ansatz entspricht im Wesentlichen der zuvor diskutierten Alternative 1. Der einzige Unterschied besteht darin, dass sich die „Positionierungs-Flexibilität“ hier auf die XML-Spalten *mehrerer* Anfrageergebnisse erstreckt. Diese Flexibilitätssteigerung stellt allerdings keinen tatsächlichen Vorteil dar: Bei einer gleichzeitigen Verarbeitung mehrerer Anfrageergebnisse werden i. d. R. in allen diesen Anfrageergebnissen *zugleich* Sequenzcursor benötigt, sodass ohnehin für jedes Anfrageergebnis eigene Sequenzcursor gebraucht werden.

  Im Vergleich zur Alternative 1 besitzen die Positionierungsanweisungen hier eine (noch) höhere Komplexität: Bei der Positionierung eines (nicht an ein Anfrageergebnis gebundenen) Sequenzcursors muss nun auch der Name des entsprechenden Tupelcursors angegeben werden. Beim Durchführen der Positionierung muss dann

zusätzlich geprüft werden, ob überhaupt ein entsprechender Tupelcursor existiert und ob dieser geöffnet ist.

- *Alternative 3: Bindung an eine Basistabelle bzw. an eine Spalte davon*

  Beim Erzeugen eines Sequenzcursors wird dieser nicht an eine Spalte vom Anfrageergebnis, sondern an eine komplette Basistabelle bzw. an eine Spalte einer Basistabelle gebunden.

  Auf ein und derselben Basistabelle können mehrere Anfrageergebnisse beruhen. Jedes dieser Anfrageergebnisse kann mehrere XML-Spalten besitzen, die alle auf ein und derselben XML-Spalte der Basistabelle basieren. Somit kann aus der Bindung an eine Spalte der Basistabelle keine Bindung an eine bestimmte Ergebnisspalte abgeleitet werden. Dies gilt erst recht, wenn eine Bindung an die gesamte Basistabelle vorliegt.

  Die Bindung eines Sequenzcursors an eine Basistabelle (bzw. an eine Spalte davon) hat also u. U. denselben Effekt wie ein kompletter Verzicht auf eine Bindung. Das hier beschriebene Vorgehen besitzt damit keinerlei Vorteile gegenüber Alternative 2.

5.1.2 Dynamische Erzeugbarkeit

Sequenzcursor können dynamisch zur Laufzeit erzeugt (und gelöscht) werden [Jat07]. Die Anzahl der Sequenzcursor muss zur Vorübersetzungszeit also noch nicht feststehen. Damit wird es möglich, die genaue Gestalt der Sequenzausschnitte (von der die Anzahl der benötigten Sequenzcursor abhängt) erst zur Laufzeit festzulegen: Ergibt sich zur Laufzeit beispielsweise, dass jeder Sequenzausschnitt jeweils nur einen einzelnen Baum umfassen soll, genügt es, *einen* Sequenzcursor zu erzeugen. Sollen hingegen mehrere direkt aufeinanderfolgende Bäume in den Sequenzausschnitten enthalten sein, ist die Erzeugung von *zwei* Sequenzcursorn erforderlich. Wird beabsichtigt, einzelne Teilbäume dieser direkt aufeinanderfolgenden Bäume auszuschließen, werden weitere Sequenzcursor gebraucht.

Das dynamische Erzeugen von Sequenzcursorn gewährleistet also ein sehr hohes Maß an Flexibilität. Diesem Vorteil steht der Nachteil gegenüber, dass es nicht möglich ist, Fehler bereits zur Vorübersetzungszeit abzufangen. So lässt sich zur Vorübersetzungszeit beispielsweise nicht testen, ob der Tupelcursor, auf dem beim Erzeugen des Sequenzcursors Bezug genommen wird, überhaupt existiert — das Erzeugen des Sequenzcursors (und damit die Bezugnahme auf den Tupelcursor) findet ja erst zur Laufzeit statt.

Alternativ zur dynamischen Sequenzcursorerzeugung wäre es denkbar, Sequenzcursor (statisch) im Quellcode zu deklarieren [Böh07]. Die fest in den Quellcode verdrahteten Sequenzcursordeklarationen könnten dann bei der Vorübersetzung auf Fehler überprüft werden. Allerdings wären dabei nur relativ unbedeutende Überprüfungen möglich. Insbesondere ließe sich (dynamisches Embedded SQL vorausgesetzt) noch nicht einmal feststellen, ob das Anfrageergebnis, dem der Sequenzcursor zugeordnet werden soll, überhaupt eine XML-Spalte besitzt: Die SQL-`SELECT`-Anfrage kann ja flexibel zur Laufzeit zusammengebaut werden, sodass die Struktur des Anfrageergebnisses zur Vorübersetzungszeit noch vollkommen unbekannt ist (vgl. Abschnitt 2.4.2).

Gegen ein statisches Deklarieren von Sequenzcursorn spricht vor allem die damit verbundene große Inflexibilität: Die Anzahl der Sequenzcursor müsste zwangsläufig bereits zur

Vorübersetzungszeit feststehen. Damit wären die Möglichkeiten, mit denen sich Sequenzausschnitte zur Laufzeit definieren lassen, deutlich eingeschränkt. Würden beispielsweise nur zwei Sequenzcursor existieren, so ließen sich keine Sequenzausschnitte definieren, die aus mehreren Bäumen bestehen, bei denen jeweils gewisse Teilbäume ausgeschlossen sind.

5.1.3 Keine Default-Positionierung

Sequenzcursor werden bei ihrer Erzeugung nicht automatisch positioniert, sondern besitzen zunächst den Status[1] *„unpositioniert“*. Auch wenn der (den Sequenzcursorn zugeordnete) Tupelcursor weiterbewegt wird, findet keine automatische Positionierung der Sequenzcursor statt [Jat07]. Die entsprechenden Sequenzcursor haben anschließend ebenfalls den Status *„unpositioniert“*.[2]

Alternativ wäre es denkbar, (beim Erzeugen von Sequenzcursorn bzw. beim Weiterbewegen des Tupelcursors) eine automatische Positionierung der Sequenzcursor durchzuführen. Das Problem bestände dabei allerdings darin, dass eine solche automatische Positionierung nicht in allen Fällen möglich wäre. Insbesondere in den folgenden Situationen könnten die Sequenzcursor nicht erfolgreich positioniert werden:

- Bei dem Spaltenwert, innerhalb dessen die Sequenzcursor eigentlich positioniert werden müssten, handelt es sich um eine leere Sequenz oder um den SQL-`NULL`-Wert.
- Der Tupelcursor steht (unmittelbar nach dem Öffnen) *vor* (und noch nicht *auf*) dem ersten Ergebnistupel. Somit existiert gar kein aktuelles Ergebnistupel.

Die erste der beiden zuvor aufgeführten Situationen kann sowohl auftreten, wenn der Tupelcursor weiterbewegt wird, als auch, wenn Sequenzcursor erzeugt werden, während der Tupelcursor auf einem Ergebnistupel positioniert ist. Die zweite Situation tritt auf, wenn Sequenzcursor erzeugt werden, *bevor* der Tupelcursor das erste Mal weiterbewegt wurde.

Eine automatische Positionierung ist also nicht immer möglich. Als naheliegender „Ausweg“ würde sich damit folgendes Vorgehen anbieten: *Sofern eine automatische Positionierung (z. B. auf den Wurzelknoten des linkesten Baums) möglich ist, wird diese auch vorgenommen. Andernfalls erhält der entsprechende Sequenzcursor den Status „unpositioniert“.*

Der Nachteil dieses Vorgehens würde aber darin bestehen, dass beim Erzeugen von Sequenzcursorn und beim Weiterbewegen des Tupelcursors nicht von vornherein klar wäre, in welchem Status sich die Sequenzcursor im Anschluss befinden. Abhängig von der Tupelcursorposition bzw. den konkreten Spaltenwerten könnten die Sequenzcursor entweder auf einen Knoten positioniert sein oder den Status *„unpositioniert“* besitzen. Somit müsste das Anwendungsprogramm vor der weiteren Nutzung der betreffenden Sequenzcursor stets explizit überprüfen, ob deren automatische Positionierung erfolgreich war oder nicht. Das hier betrachtete Vorgehen würde also nicht zu einer Vereinfachung des Anwendungsprogramms (bzw. des Programmablaufs) führen.

[1] Wir werden im Abschnitt 5.4 näher darauf eingehen, welche (weiteren) Möglichkeiten es für den Status eines Sequenzcursors gibt.

[2] Da Sequenzcursor stets nur innerhalb des jeweils aktuellen Ergebnistupels positioniert sein dürfen, können sie ihre Positionen nicht beibehalten, wenn der Tupelcursor weiterbewegt wird.

5.1.4 Typed-value-orientierte Repräsentation als Basis der Positionierung

Wie bereits in Abschnitt 4.1 ausgeführt, basiert die Positionierung der Sequenzcursor auf der im Kapitel 3 eingeführten typed-value-orientierten Repräsentation von Sequenzen. Dies bedeutet, dass die Knoten der in typed-value-orientierter Repräsentation vorliegenden Sequenzen als „Navigationsgranulat“ [MR05, Rab05] bzw. „Positionierungsgranulat“ dienen: Die Sequenzcursor können prinzipiell auf jeden einzelnen dieser Knoten positioniert werden. Eine Navigation *innerhalb* eines Knotens (also ein „Eintauchen“ in einen Knoten) ist hingegen *nicht* möglich.

Die Alternative zur Nutzung der typed-value-orientierten Repräsentation bestände darin, die Positionierung der Sequenzcursor auf der Grundlage der traditionellen Sequenzrepräsentation durchzuführen. Wie im Abschnitt 3.1.1 ausführlich geschildert, wären damit aber erhebliche Nachteile verbunden.

5.2 Positionierungsmöglichkeiten

Im Abschnitt 5.2.1 erläutern wir zunächst, warum bei der Positionierung eines Sequenzcursors nicht explizit auf eine bestimmte Sequenz Bezug genommen wird. Anschließend stellen wir im Abschnitt 5.2.2 die grundlegenden Positionierungsmöglichkeiten vor. Bei der Positionierung eines Sequenzcursors können die Knotenart, die Knotentypen und die Knotennamen berücksichtigt werden. Wir gehen hierauf in den Abschnitten 5.2.3, 5.2.4 bzw. 5.2.5 näher ein. Eine *gleichzeitige* Berücksichtigung von Knotenart, Knotentyp und Knotenname ist ebenfalls möglich. Nähere Erläuterungen hierzu enthält Abschnitt 5.2.6. Zum Abschluss befasst sich Abschnitt 5.2.7 damit, dass im Rahmen der Sequenzcursorbasierten Verarbeitung auf weitergehende Positionierungsmöglichkeiten für Sequenzcursor verzichtet wird.

5.2.1 Keine explizite Sequenzauswahl

Ein Sequenzcursor ist an eine bestimmte Spalte eines konkreten Anfrageergebnisses gebunden (Abschnitt 5.1.1). Außerdem darf ein Sequenzcursor nur im jeweils aktuellen Ergebnistupel positioniert werden — also in dem Ergebnistupel, auf dem der (dem Sequenzcursor zugeordnete) Tupelcursor aktuell steht. Somit ist für jeden Sequenzcursor zu jedem Zeitpunkt jeweils eindeutig bestimmt, innerhalb welchen Spaltenwerts — d. h. innerhalb welcher Sequenz — eine Positionierung möglich ist (vgl. *Abbildung 5.1*).[3] Bei der Positionierung eines Sequenzcursors wird deshalb nicht explizit auf eine bestimmte Sequenz Bezug genommen. Dies bedeutet, dass in der Sequenzcursoranweisung `MOVE SEQUENCE CURSOR` (Abschnitt 7.2.3) weder eine Ergebnisspalte noch eine Ergebniszeile spezifiziert wird.

[3] Sofern es sich bei dem entsprechenden Spaltenwert allerdings um eine leere Sequenz oder um den SQL-`NULL`-Wert handelt, ist (momentan) *keine* Positionierung des Sequenzcursors möglich. Gleiches gilt für den Fall, dass der (dem Sequenzcursor zugeordnete) Tupelcursor nicht auf einem Ergebnistupel steht.

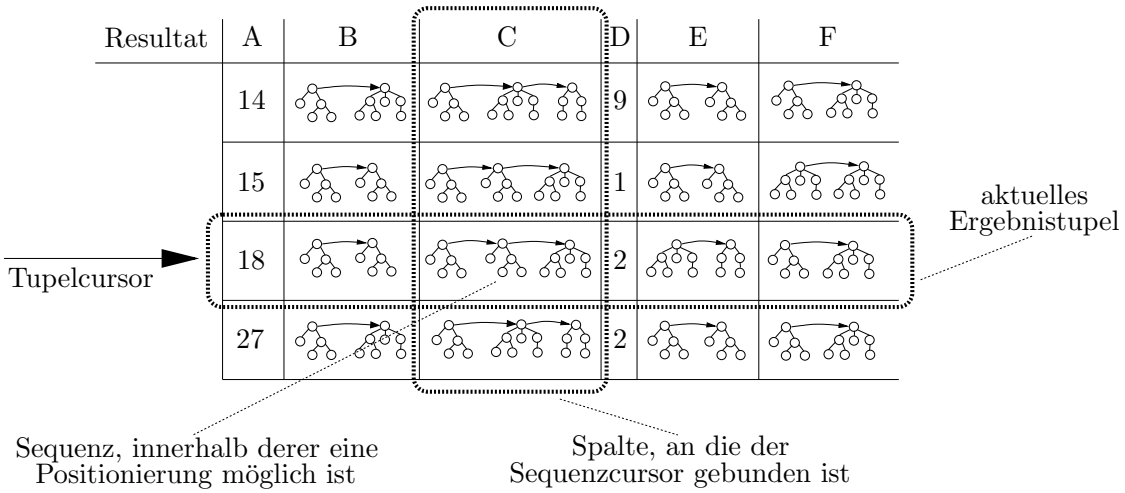


Abbildung 5.1: Bestimmung der für die Positionierung relevanten Sequenz

5.2.2 Grundlegende Positionierungsmöglichkeiten

Im aktuellen Abschnitt beschreiben wir, wie Sequenzcursor *ohne* Berücksichtigung der Knotenart, des Knotentyps und des Knotennamens positioniert werden können. Aufbauend auf diesen grundlegenden Positionierungsmöglichkeiten werden wir in den Abschnitten 5.2.3 bis 5.2.6 dann erläutern, wie eine von Knotenart, Knotentyp bzw. Knotenname abhängige Positionierung möglich ist.

Bei der Positionierung von Sequenzcursorn unterscheiden wir grundsätzlich zwischen einer *absoluten Positionierung* (gezieltes Anspringen eines bestimmten Knotens unabhängig von der bisherigen Sequenzcursorposition) und einer *relativen Positionierung* (Weiterbewegen eines Sequenzcursors ausgehend von seiner aktuellen Position). In den Abschnitten 5.2.2.1 und 5.2.2.2 stellen wir die grundlegenden Möglichkeiten für eine absolute bzw. relative Positionierung vor.

5.2.2.1 Absolute Positionierung

Gemäß unseres Sprachvorschlags (Kapitel 7) existieren für eine absolute Positionierung von Sequenzcursorn die im Folgenden aufgeführten Möglichkeiten [Böh07]. Für nähere Details bezüglich der in den Syntaxbeispielen genutzten Sequenzcursoranweisung `MOVE SEQUENCE CURSOR` verweisen wir auf Abschnitt 7.2.3.

- Die Sequenzcursor können gezielt auf den Wurzelknoten des ersten bzw. letzten Baums positioniert werden *(Abbildung 5.2)*. Von dieser Möglichkeit haben wir bereits in unserem Beispielszenario Gebrauch gemacht (Abschnitt 4.2.4).

```
MOVE meinSC TO ROOT OF FIRST TREE;
MOVE meinSC TO ROOT OF LAST TREE;
```

Abbildung 5.2: Absolute Positionierung von Sequenzcursorn — Syntaxbeispiel 1

- Eine Positionierung auf die Wurzelknoten der restlichen Bäume ist ebenfalls in einem Schritt möglich. So kann beispielsweise die Wurzel des siebenten Baums direkt angesprungen werden *(Abbildung 5.3)*. Bezogen auf unser Beispielszenario ist es damit möglich, einen Sequenzcursor in einem Schritt auf das siebente Bonusangebot (oder ein beliebiges anderes Bonusangebot) zu positionieren.

```
MOVE meinSC TO ROOT OF TREE NUMBER 7;
```

Abbildung 5.3: Absolute Positionierung von Sequenzcursorn — Syntaxbeispiel 2

- Die Position eines anderen (bereits positionierten) Sequenzcursors kann gezielt angesprungen werden *(Abbildung 5.4)*. Dies ist z. B. hilfreich, wenn bei der Sequenzausschnittsdefinition ein gewisser Teilbaum eines Baums ausgeschlossen werden soll: Hierzu wird zunächst ein Sequenzcursor (mittels absoluter oder relativer Positionierungsanweisungen) auf die Wurzel des entsprechenden Baums bewegt. Mit einem zweiten Sequenzcursor wird dann die Position des ersten Sequenzcursors gezielt angesprungen. Anschließend wird der zweite Sequenzcursor (mit Hilfe relativer Positionierungsanweisungen) im Baum nach unten bewegt und dabei auf die Wurzel des auszuschließenden Teilbaums positioniert.

```
MOVE meinSC TO POSITION OF andererSC;
```

Abbildung 5.4: Absolute Positionierung von Sequenzcursorn — Syntaxbeispiel 3

5.2.2.2 Relative Positionierung

Unser Sprachvorschlag sieht für eine relative Positionierung [Ott07] von Sequenzcursorn die folgenden Möglichkeiten vor (vgl. Abschnitt 7.2.3):

- Ein auf dem Wurzelknoten eines Baums positionierter Sequenzcursor kann zum Wurzelknoten des rechten bzw. linken Nachbarbaums bewegt werden *(Abbildung 5.5)*. Durch eine wiederholte Ausführung der entsprechenden Positionierungsanweisung lässt sich die Sequenz auf Wurzelebene (von links nach rechts bzw. von rechts nach links) Baum für Baum durchlaufen.[4]

```
MOVE meinSC TO ROOT OF NEXT TREE;
MOVE meinSC TO ROOT OF PREVIOUS TREE;
```

Abbildung 5.5: Relative Positionierung von Sequenzcursorn — Syntaxbeispiel 1

- Die Sequenzcursor können auf der Wurzelebene der Sequenz um eine beliebige Schrittweite nach rechts bzw. links bewegt werden. So ist z. B. ein Weiterbewegen um 21 Schritte nach rechts oder um fünf Schritte nach links möglich *(Abbildung 5.6)*. Diese Art der Positionierung haben wir bereits in unserem Beispielszenario genutzt (Abschnitt 4.2.4).

[4] Für ein komplettes Durchlaufen *aller* Wurzelknoten muss der Sequenzcursor zu Beginn auf der Wurzel des ersten bzw. letzten Baums stehen.

```
MOVE meinSC FORWARD 21 STEPS;
MOVE meinSC BACKWARD 5 STEPS;
```

Abbildung 5.6: Relative Positionierung von Sequenzcursorn — Syntaxbeispiel 2

- Ein Sequenzcursor, der auf einem Knoten positioniert ist, kann zum ersten (d. h. linkesten) bzw. letzten (d. h. rechtesten) Kindknoten des entsprechenden Knotens bewegt werden *(Abbildung 5.7)*. Damit wird es möglich, „nach unten“ in die Bäume „einzutauchen“. Ein solches Eintauchen ist erforderlich, um bei der Festlegung von Sequenzausschnitten gewisse Teilbäume ausschließen zu können.

```
MOVE meinSC TO FIRST CHILD;
MOVE meinSC TO LAST CHILD;
```

Abbildung 5.7: Relative Positionierung von Sequenzcursorn — Syntaxbeispiel 3

  Wir möchten an dieser Stelle nochmals ausdrücklich darauf hinweisen, dass es sich bei einem A-Knoten, der einem E-Knoten zugeordnet ist, *nicht* um dessen Kindknoten handelt (vgl. Abschnitt 2.10.2.2).[5] Die in Abbildung 5.7 gezeigten Anweisungen sind somit nicht dafür geeignet, einen Sequenzcursor auf einen A-Knoten zu positionieren. Wir werden im Folgenden beschreiben, wie ein Sequenzcursor auf einen (einem E-Knoten zugeordneten) A-Knoten positioniert werden kann.

- Ein Sequenzcursor, der auf einem E-Knoten positioniert ist, kann zum ersten (d. h. linkesten) bzw. letzten (d. h. rechtesten) der diesem E-Knoten zugeordneten A-Knoten bewegt werden *(Abbildung 5.8)*. Wie bei der zuvor betrachteten Positionierung auf den ersten bzw. letzten Kindknoten handelt es sich hierbei um eine Möglichkeit, nach unten in die Bäume einzutauchen.

```
MOVE meinSC TO FIRST ATTRIBUTE;
MOVE meinSC TO LAST ATTRIBUTE;
```

Abbildung 5.8: Relative Positionierung von Sequenzcursorn — Syntaxbeispiel 4

- Ein Sequenzcursor, der auf einem Knoten positioniert ist, kann nicht nur zum ersten und letzten Kindknoten bewegt werden, sondern auch zu jedem der dazwischenliegenden Kindknoten. Dies gilt analog für eine Positionierung auf die dem Knoten zugeordneten A-Knoten. So lässt sich der Sequenzcursor in einem Schritt beispielsweise zum zweiten Kindknoten oder zum achten A-Knoten weiterbewegen *(Abbildung 5.9)*.

```
MOVE meinSC TO CHILD NUMBER 2;
MOVE meinSC TO ATTRIBUTE NUMBER 8;
```

Abbildung 5.9: Relative Positionierung von Sequenzcursorn — Syntaxbeispiel 5

[5] In der traditionellen Sequenzrepräsentation spielen N-Knoten bezüglich der Vater-Kind-Beziehungen eine ähnliche Sonderrolle wie A-Knoten. Die Positionierung der Sequenzcursor basiert jedoch nicht auf der traditionellen Repräsentation von Sequenzen, sondern auf der typed-value-orientierten Repräsentation. Da bei letzterer Repräsentationsvariante keine N-Knoten existieren (Abschnitt 3.8.1), erübrigt sich hier eine spezielle Betrachtung/Behandlung von N-Knoten.

- Ein Sequenzcursor, der nicht auf der Wurzelebene der Sequenz positioniert ist, kann zum nächstgelegenen rechten bzw. linken Geschwisterknoten bewegt werden *(Abbildung 5.10)*. Als Geschwisterknoten gelten dabei zum einen alle A-Knoten, die ein und demselben E-Knoten zugeordnet sind. Zum anderen werden alle Kindknoten ein und desselben Knotens als Geschwister aufgefasst. Zwischen einem A-Knoten und einem Knoten einer anderen Knotenart kann keine Geschwisterbeziehung bestehen.[6]

  Durch ein wiederholtes Ausführen dieser Positionierungsanweisungen lassen sich alle Kindknoten eines bestimmten Knotens bzw. alle diesem Knoten zugeordneten A-Knoten Schritt für Schritt durchlaufen.

```
MOVE meinSC TO NEXT SIBLING;
MOVE meinSC TO PREVIOUS SIBLING;
```

Abbildung 5.10: Relative Positionierung von Sequenzcursorn — Syntaxbeispiel 6

- Ein Sequenzcursor, der nicht auf der Wurzelebene der Sequenz positioniert ist, lässt sich mit Hilfe der ersten in *Abbildung 5.11* gezeigten Anweisung zum Vaterknoten bewegen. Durch mehrfache Ausführung dieser Positionierungsanweisung kann bis zum Wurzelknoten des entsprechenden Baums navigiert werden. Alternativ ist es möglich, den Sequenzcursor in einem Schritt zum Wurzelknoten zu bewegen. Hierfür steht die zweite in Abbildung 5.11 dargestellte Anweisung zur Verfügung.

  Die hier betrachteten Anweisungen ermöglichen es also, innerhalb der Bäume (wieder) „aufzusteigen“ — sie bilden damit das Gegenstück zu jenen Anweisungen, mittels derer nach unten in die Bäume eingetaucht werden kann.

```
MOVE meinSC TO PARENT;
MOVE meinSC TO ROOT OF CURRENT TREE;
```

Abbildung 5.11: Relative Positionierung von Sequenzcursorn — Syntaxbeispiel 7

5.2.3 Berücksichtigung der Knotenart

Entsprechend unseres Sprachvorschlags (Kapitel 7) kann bei der Positionierung von Sequenzcursorn die Art der Knoten berücksichtigt werden [Böh07]. Dass dies auch tatsächlich sinnvoll ist, soll folgendes kleines Beispiel zeigen: Eine Sequenz könnte mit Hilfe von C-Knoten in verschiedene logische Abschnitte unterteilt sein. Es wäre dann hilfreich, mit einem Sequenzcursor direkt zum nächsten C-Knoten (also zum Anfang des nächsten logischen Sequenzabschnitts) springen zu können.

Mittels der in *Abbildung 5.12* dargestellten Anweisung lässt sich ein auf der Wurzelebene der Sequenz positionierter Sequenzcursor namens *meinSC* (wie im zuvor betrachteten Beispiel erwünscht) in einem Schritt zum nächsten in der Wurzelebene enthaltenen C-Knoten weiterbewegen. Die Positionierungsanweisung `MOVE meinSC TO ROOT OF NEXT TREE` aus Abschnitt 5.2.2.2 wurde hierzu einfach um die Angabe der Knotenart erweitert.

[6] Die Definition der Geschwisterbeziehung wird hier dadurch verkompliziert, dass neben traditionellen Vater-Kind-Beziehungen auch einseitige Vater-Kind-Beziehungen möglich sind.

```
MOVE meinSC TO ROOT OF NEXT TREE OF KIND COMMENT;
```

Abbildung 5.12: Berücksichtigung der Knotenart — Syntaxbeispiel 1

Die typed-value-orientierte Sequenzrepräsentation (welche als Grundlage für die Navigation der Sequenzcursor dient) unterscheidet die folgenden Knotenarten: *D (Document)*, *E (Element)*, *A (Attribute)*, *P (Processing Instruction)*, *C (Comment)*, *T (Text)* und *V (Atomic Value)*. Damit stehen für eine Angabe der Knotenart diese sieben Möglichkeiten zur Verfügung.

Auch andere der im Abschnitt 5.2.2 aufgeführten Positionierungsanweisungen lassen sich um die Angabe einer Knotenart erweitern. So ist es beispielsweise möglich, zum zweitlinkesten (sich in der Wurzelebene befindenden) E-Knoten zu springen oder den rechtesten T-Kindknoten anzusteuern *(Abbildung 5.13)*. Für nähere Details bezüglich Syntax und Semantik verweisen wir auf Abschnitt 7.2.3.

```
MOVE meinSC TO ROOT OF TREE NUMBER 2 OF KIND ELEMENT;
MOVE meinSC TO LAST CHILD OF KIND TEXT;
```

Abbildung 5.13: Berücksichtigung der Knotenart — Syntaxbeispiel 2

Es existieren jedoch auch absolute und relative Positionierungsanweisungen, bei denen eine Erweiterung um die Angabe einer Knotenart nicht sinnvoll wäre. Hierzu zählen z. B. die Anweisungen `MOVE meinSC TO POSITION OF andererSC`, `MOVE meinSC TO FIRST ATTRIBUTE` und `MOVE meinSC TO PARENT`. Weitere Einzelheiten zu dieser Thematik enthält Abschnitt 7.2.3.

5.2.4 Berücksichtigung des Knotentyps

Ähnlich wie die Berücksichtigung der Knotenart kann bei der Sequenzcursorpositionierung auch die Berücksichtigung des Knotentyps sinnvoll sein [Böh07]: Eine Sequenz könnte auf der Wurzelebene E-Knoten verschiedenen Typs enthalten. So wäre es denkbar, dass außer E-Knoten vom Typ *BonusAngebotsTyp* (vgl. Abschnitt 3.9) auch E-Knoten anderer Typen vorkommen. Für die Festlegung des (die aufwertbaren Bonusangebote umfassenden) Sequenzausschnitts wäre es dann u. a. hilfreich, mit dem Sequenzcursor direkt zum letzten sich auf Wurzelebene befindenden Knoten vom Typ *BonusAngebotsTyp* (also zum letzten Bonusangebot) springen zu können.

Ein direktes Anspringen des entsprechenden Knotens ist mit Hilfe der in *Abbildung 5.14* gezeigten Anweisung möglich. Die aus dem Abschnitt 5.2.2.1 bekannte Positionierungsanweisung `MOVE meinSC TO ROOT OF LAST TREE` wurde hierzu um die Angabe des Knotentyps erweitert.

```
MOVE meinSC TO ROOT OF LAST TREE OF TYPE BonusAngebotsTyp;
```

Abbildung 5.14: Berücksichtigung des Knotentyps — Syntaxbeispiel

Analog zur im vorigen Abschnitt betrachteten Berücksichtigung der Knotenart ist es möglich, einige — jedoch nicht alle — der im Abschnitt 5.2.2 vorgestellten Anweisun-

gen um eine Angabe des Knotentyps zu erweitern. Nähere Details dazu sowie zur Syntax und Semantik der zulässigen Positionierungsanweisungen finden sich im Abschnitt 7.2.3.

5.2.5 Berücksichtigung des Knotennamens

Bei der Positionierung der Sequenzcursor können auch die Knotennamen berücksichtigt werden [Jat07]. Dies ist beispielsweise in folgender Situation sinnvoll: Eine Sequenz könnte auf der Wurzelebene sowohl E-Knoten mit dem Namen *BonusAngebot* als auch E-Knoten namens *FestesBonusAngebot* enthalten. Die E-Knoten mit Namen *FestesBonusAngebot* (die wie die E-Knoten namens *BonusAngebot* den Typ *BonusAngebotsTyp* besitzen) sollen dabei Bonusangebote repräsentieren, für die eine Aufwertung generell ausgeschlossen ist. Für die Festlegung des Sequenzausschnitts wäre es dann u. a. hilfreich, den Sequenzcursor direkt auf das letzte aufwertbare Bonusangebot (also den letzten in der Wurzelebene enthaltenen Knoten namens *BonusAngebot*) positionieren zu können.[7] Eine solche Positionierung ist gemäß unseres Sprachvorschlags mit Hilfe der in *Abbildung 5.15* dargestellten Anweisung möglich.

```
MOVE meinSC TO ROOT OF LAST TREE OF NAME BonusAngebot;
```

Abbildung 5.15: Berücksichtigung des Knotennamens — Syntaxbeispiel

Ähnlich wie bei der Berücksichtigung von Knotenart bzw. Knotentyp können bestimmte (aber nicht alle) der im Abschnitt 5.2.2 aufgeführten Anweisungen um die Angabe eines Knotennamens erweitert werden. Weitere Einzelheiten hierzu sowie zur Syntax und Semantik enthält Abschnitt 7.2.3.

5.2.6 Gleichzeitige Berücksichtigung von Knotenart, -typ und -name

Wie das folgende Beispiel zeigt, gibt es Situationen, in denen es bei der Positionierung von Sequenzcursorn nicht ausreicht, entweder nur die Knotenart, nur den Knotentyp oder nur den Knotennamen zu berücksichtigen. Aus diesem Grund erlaubt unser Sprachvorschlag eine *gleichzeitige* Berücksichtigung von Knotenart, Knotentyp und Knotenname [Böh07].

Eine Sequenz enthalte auf der Wurzelebene sowohl E-Knoten namens *Preis* und *Rabatt* als auch A-Knoten mit den Namen *Preis* und *Rabatt*. Soll ein Sequenzcursor nun auf den linkesten E-Knoten namens *Preis* positioniert werden, so genügt dabei weder die alleinige Berücksichtigung des Knotennamens (es gibt ja auch A-Knoten mit dem Namen *Preis*) noch die alleinige Berücksichtigung der Knotenart (es existieren ja auch E-Knoten mit anderem Namen). Es ist hier also notwendig, die Knotenart und den Knotennamen *gemeinsam* zu berücksichtigen.

Abbildung 5.16 zeigt die in unserem Beispiel zu nutzende Positionierungsanweisung. Es handelt sich dabei um eine Abwandlung der Anweisung `MOVE meinSC TO ROOT OF FIRST TREE` aus Abschnitt 5.2.2.1. Diese Anweisung wurde sowohl um die Angabe einer Knotenart als auch um die Angabe eines Knotennamens erweitert. Für nähere Details zu Syntax und Semantik verweisen wir auf Abschnitt 7.2.3

[7] Da alle Wurzelknoten dieselbe Art *(E)* und denselben Typ *(BonusAngebotsTyp)* besitzen, reicht eine Berücksichtigung von Knotenart und Knotentyp hier nicht aus.

```
MOVE meinSC TO ROOT OF FIRST TREE OF KIND ELEMENT OF NAME Preis;
```

Abbildung 5.16: Berücksichtigung von Knotenart und Knotenname — Syntaxbeispiel

Es ist zu beachten, dass die gleichzeitige Angabe von Knotenart, Knotentyp und Knotenname gewissen Beschränkungen unterliegt. So ist es beispielsweise nicht zulässig, einen Knotentyp oder Knotennamen zu spezifizieren, wenn zugleich die Knotenart *Comment* gefordert wird — C-Knoten besitzen ja weder einen Typ noch einen Namen (vgl. Abschnitt 2.10.2.4 bzw. 2.10.2.3). Weitere Einzelheiten hierzu finden sich im Abschnitt 7.2.3.

5.2.7 Verzicht auf weitergehende Positionierungsmöglichkeiten

Die in unserem Sprachvorschlag vorgesehenen Positionierungsanweisungen reichen für das Beispielszenario *Kundenkartenverwaltung* (und für eine Vielzahl anderer Anwendungsszenarien) vollkommen aus. Natürlich sind prinzipiell auch darüber hinausgehende („beliebig komplexe“) Positionierungsmöglichkeiten denkbar. So könnten beispielsweise Positionierungsanweisungen konzipiert werden, die selbst folgender (in irgendeinem denkbaren Anwendungsszenario vielleicht durchaus sinnvollen) Positionierungsanforderung gerecht werden: *Positioniere den Sequenzcursor auf den linkesten in der Wurzelebene der Sequenz enthaltenen E-Knoten, der zwischen zwei und acht zugeordnete A-Knoten besitzt und zudem mindestens sieben Kindknoten hat, von denen mehr als die Hälfte E-Knoten sind, die ihrerseits jeweils entweder genau drei C-Kindknoten oder mindestens fünf (dreistellige Primzahlen repräsentierende) V-Kindknoten haben.* Gegen die Einführung derart mächtiger Positionierungsanweisungen spricht jedoch die damit verbundene Komplexität.

Wir möchten an dieser Stelle nochmals ausdrücklich hervorheben, dass die Sequenzcursorpositionierung innerhalb von Sequenzen erfolgt, die im *Ergebnis* einer SQL-`SELECT`-Anfrage enthalten sind. Die Navigation der Sequenzcursor findet also auf einem *Anfrageergebnis* statt. Somit ist es weder sinnvoll noch notwendig, Positionierungsanweisungen bereitzustellen, die die Funktionalität bzw. Ausdruckskraft einer vollwertigen Anfragesprache besitzen. Aus diesem Grund ist es in unserem Sprachentwurf auch nicht vorgesehen, die Positionierung der Sequenzcursor mit Hilfe von XPath-Ausdrücken oder XQuery-Anfragen vorzunehmen.[8]

Dass die von uns vorgeschlagenen Positionierungsanweisungen in ihrer Komplexität und Funktionalität begrenzt sind, ist übrigens konform zu den allgemeinen „SQL-Sprachgewohnheiten“: Ein scrollable SQL-Tupelcursor kann auf dem Anfrageergebnis zwar um eine beliebige Schrittweite nach vorn oder hinten bewegt werden — weitergehende (komplexere) Positionierungsmöglichkeiten (wie etwa die Nutzung von Suchbedingungen) werden jedoch nicht angeboten.

5.3 Erfolg bzw. Misserfolg der Positionierung

Die Ausführung einer Positionierungsanweisung kann erfolgreich verlaufen oder scheitern. Wir werden in den Abschnitten 5.3.1 und 5.3.2 näher auf diese beiden Fälle eingehen.

[8] Da die Positionierung der Sequenzcursor auf der typed-value-orientierten Repräsentation basiert, könnten XPath-Ausdrücke und XQuery-Anfragen ohnehin nicht ohne weiteres für die Sequenzcursorpositionierung genutzt werden.

Die Sequenzcursoranweisungen — zu denen auch die Positionierungsanweisungen gehören — wurden von uns als SQL-Erweiterung konzipiert. Zum Testen, ob die Ausführung einer Positionierungsanweisung erfolgreich war oder nicht, können somit die herkömmlichen (von SQL bereitgestellten) Mechanismen genutzt werden: Auskunft über Erfolg bzw. Misserfolg einer Sequenzcursorpositionierung geben die `SQLSTATE`- und `SQLCODE`-Werte, die nach der Anweisungsausführung vom SQL-Laufzeitsystem zurückgeliefert werden (vgl. Abschnitt 2.3.2).

5.3.1 Erfolgreicher Verlauf der Positionierung

Im Abschnitt 5.3.1.1 erläutern wir zunächst, dass das Feedback, welches ein Anwendungsprogramm nach einer erfolgreichen Sequenzcursorpositionierung vom SQL-Laufzeitsystem erhält, über die bloße Meldung der fehlerfreien Anweisungsausführung hinausgeht. Dass dadurch der Begriff *Sequenzcursor* gerechtfertigt wird, ist anschließend Gegenstand von Abschnitt 5.3.1.2. Die Informationen, die nach einer erfolgreichen Ausführung einer Positionierungsanweisung vom SQL-Laufzeitsystem zurückgeliefert werden, können vom Anwendungsprogramm auch abgefragt werden, ohne dass dazu ein Sequenzcursor bewegt werden muss. Nähere Details hierzu enthält Abschnitt 5.3.1.3.

5.3.1.1 Feedback bei erfolgreicher Positionierung

Nach der erfolgreichen Positionierung eines Sequenzcursors werden dem Anwendungsprogramm gewisse Informationen über den Knoten zur Verfügung gestellt, auf dem der Sequenzcursor nun steht. Hierzu zählen u. a. die Art, (und sofern vorhanden) der Typ und der Name des entsprechenden Knotens. Dass ein solches Feedback tatsächlich sinnvoll ist, sollen folgende Beispiele zeigen, bei denen es sich um leichte Abwandlungen bzw. Erweiterungen des Anwendungsszenarios *Kundenkartenverwaltung* handelt.

- Manche Bonusangebotesequenzen enthalten am Ende einen C-Knoten, der die Aufwertbarkeit der Bonusangebote einschränkt: Ein C-Knoten an hinterster Position signalisiert, dass nur die letzten drei statt der letzten fünf Bonusangebote im Anwendungsprogramm aufgewertet werden dürfen.

  Für die Festlegung des Sequenzausschnitts ist hier also die Art des letzten in der Wurzelebene der Sequenz enthaltenen Knotens von Interesse. Es wäre somit hilfreich, wenn einfach ein Sequenzcursor auf diesen Knoten positioniert werden könnte und sich anschließend eine Abfrage der Knotenart durchführen ließe.

- Die Bonusangebotesequenzen können auf der Wurzelebene E-Knoten verschiedenen Typs enthalten. Bei den einzelnen Bäumen sollen dann — abhängig vom Typ des jeweiligen Wurzelknotens — gewisse Teilbäume ausgeschlossen werden (d. h., die entsprechenden Teilbäume sollen nicht zum Sequenzausschnitt gehören).

  Es wäre hier also von Vorteil, wenn beim Durchlaufen der Wurzelknoten jeweils ermittelt werden könnte, welchen Typ die einzelnen Knoten haben.

- Auf der Wurzelebene von Bonusangebotesequenzen können außer E-Knoten namens *BonusAngebot* auch E-Knoten mit anderen Namen vorkommen. Bei Bäumen, die einen Wurzelknoten mit einem von *BonusAngebot* abweichenden Namen besitzen,

sollen — in Abhängigkeit vom konkreten Namen — bestimmte Teilbäume ausgeschlossen werden.

Von Interesse sind hier also die Knotennamen. Es wäre somit vorteilhaft, wenn sich diese beim Durchlaufen der Wurzelknoten unkompliziert ermitteln ließen.

Um dem Anwendungsprogramm nach einer erfolgreichen Sequenzcursorpositionierung Informationen über den Knoten mitzuteilen, auf dem der entsprechende Sequenzcursor nun steht, wird die so genannte *SQLSCFA (SQL Sequence Cursor Feedback Area)* genutzt. Analog zur SQLCA (Abschnitt 2.3.2), zur SQLDA (Abschnitt 2.4.2) und zur SQLSPCA (Abschnitt 4.2.6) handelt es sich bei der von uns eingeführten SQLSCFA [Jat07] um eine Datenstruktur im Anwendungsprogramm, die zur Kommunikation mit dem SQL-Laufzeitsystem dient: Nach jeder erfolgreichen Ausführung einer Positionierungsanweisung vermerkt das SQL-Laufzeitsystem gewisse Informationen (z. B. die Art, den Typ sowie den Namen des Knotens, auf dem der entsprechende Sequenzcursor nach der Positionierung steht) in der SQLSCFA. Das Anwendungsprogramm kann diese Informationen anschließend aus der SQLSCFA auslesen und geeignet nutzen.

Im Gegensatz zur SQLCA, die nach *jeder* Ausführung einer *beliebigen* SQL-Anweisung aktualisiert wird, findet eine Aktualisierung der SQLSCFA nur im Anschluss an eine *erfolgreiche* Ausführung einer *Sequenzcursorpositionierungs*-Anweisung statt.[9] Für nähere Details zur SQLSCFA verweisen wir auf Abschnitt 7.4.1.

5.3.1.2 Rechtfertigung des Sequenzcursor-Begriffs

Im Rahmen der Sequenzcursor-basierten Verarbeitung werden Sequenzcursor eigentlich nicht als (Datenbank-)Cursor im klassischen Sinn verwendet, sondern eher als eine Art „Begrenzungsmarkierungen“, auf die bei der Definition der Sequenzausschnitte Bezug genommen wird. Dass der Cursor-Begriff hier aber dennoch gerechtfertigt ist, soll folgende Betrachtung zeigen:

Nach dem erfolgreichen Positionieren/Weiterbewegen eines Sequenzcursors werden verschiedene Informationen (z. B. die Knotenart, der Knotentyp und der Knotenname) in die SQLSCFA — und damit ins Anwendungsprogramm — übertragen. Somit besitzt ein Sequenzcursor die beiden wesentlichen Eigenschaften eines klassischen Cursors:

1. Der Cursor wird über das Anfrageergebnis bewegt.
2. Beim Bewegen des Cursors werden Daten ins Anwendungsprogramm übertragen.

Die von uns eingeführte Bezeichnung „Sequenz*cursor*“ ist also passend.

5.3.1.3 Informationsabfrage ohne Positionierung

Die SQLSCFA wird nach jeder erfolgreichen Sequenzcursorpositionierung aktualisiert. Die in der SQLSCFA enthaltenen Informationen beziehen sich somit grundsätzlich auf den *zuletzt* positionierten Sequenzcursor. In bestimmten Situationen kann es jedoch hilfreich

[9] Die SQLSCFA wird auch nach der Ausführung der Sequenzcursoranweisung `GET INFORMATION ABOUT SEQUENCE CURSOR` aktualisiert (vgl. Abschnitt 5.3.1.3).

sein, auch Informationen bezüglich eines anderen Sequenzcursors abrufen zu können. Dabei sollte es nicht notwendig sein, den betreffenden Sequenzcursor bewegen zu müssen — er besitzt u. U. ja bereits die gewünschte Position.

Für diesen Zweck steht die Sequenzcursoranweisung `GET INFORMATION ABOUT SEQUENCE CURSOR` zur Verfügung, welche im Abschnitt 7.2.4 detailliert vorgestellt wird. Ihre Ausführung bewirkt (wie die erfolgreiche Positionierung eines Sequenzcursors) eine Aktualisierung der SQLSCFA. Sind beispielsweise Informationen über den Sequenzcursor *meinSC* von Interesse, lassen sich diese mit Hilfe der in *Abbildung 5.17* gezeigten Anweisung ermitteln [Böh07].

```
GET INFORMATION ABOUT SEQUENCE CURSOR meinSC;
```

Abbildung 5.17: Informationsabfrage ohne Positionierung — Syntaxbeispiel

Es ist möglich, dass der zuvor betrachtete Sequenzcursor *meinSC* gar nicht auf einem Knoten positioniert ist, sondern sich im Status *„unpositioniert“* befindet. In diesem Fall könnte der Sequenzcursor (momentan) nicht zur Definition eines Sequenzausschnitts herangezogen werden — auch eine relative Positionierung des Sequenzcursors wäre (zeitweilig) ausgeschlossen.

Der aktuelle Status eines Sequenzcursors ist also sehr wichtig. Aus diesem Grund wird in der SQLSCFA (nach einer erfolgreichen Sequenzcursorpositionierung bzw. nach Ausführung der Anweisung `GET INFORMATION ABOUT SEQUENCE CURSOR`) u. a. auch der Status des entsprechenden Sequenzcursors vermerkt. Das Anwendungsprogramm kann die Statusinformation dann aus der SQLSCFA auslesen und geeignet nutzen.

Zu den Informationen, die bezüglich eines Sequenzcursors in der SQLSCFA verzeichnet werden, gehört auch eine Positionsinformation — vorausgesetzt natürlich, der Sequenzcursor befindet sich nicht gerade im Status *„unpositioniert“*. Für einen auf der Wurzelebene positionierten Sequenzcursor besagt die Positionsinformation z. B., ob der Sequenzcursor auf der Wurzel des siebenten, siebzigsten oder siebentausendsten Baums steht. Mit Hilfe der Positionsinformationen kann das Anwendungsprogramm somit beispielsweise ermitteln, ob zwei Sequenzcursor auf denselben Knoten positioniert sind oder nicht. Weitere Einzelheiten über die Positionsinformation (sowie über andere in der SQLSCFA abgelegte Informationen) finden sich im Abschnitt 7.4.1.

5.3.2 Scheitern der Positionierung

Das Fehlschlagen der Ausführung einer Anweisung zur Sequenzcursorpositionierung kann verschiedene Gründe haben [Böh07]. Einer dieser Gründe besteht darin, dass die Positionierung nicht wie gewünscht durchgeführt werden kann, weil der Knoten, auf den der Sequenzcursor eigentlich positioniert werden soll, nicht existiert.

Im Folgenden werden wir diese (Fehler-)Situation anhand eines konkreten Beispiels näher betrachten. Wir setzen hierzu eine aus sieben Bäumen bestehende Sequenz voraus, bei der ein Sequenzcursor auf die Wurzel des zweitlinkesten Baums positioniert ist. Wird nun versucht, diesen Sequenzcursor um zehn Schritte nach rechts zu bewegen, so kann diese Positionierung nicht erfolgreich durchgeführt werden — rechts von der ursprünglichen Sequenzcursorposition gibt es ja nur fünf (und nicht zehn) Wurzelknoten. Die Ausführung

der Positionierungsanweisung scheitert somit komplett. Dies bedeutet, dass der Sequenzcursor sowohl seine bisherige Position als auch seinen bisherigen Status beibehält — er ist also weiterhin auf den zweitlinkesten Wurzelknoten positioniert.

Abweichend von dem zuvor beschriebenen Verhalten wären die folgenden (im Rahmen der Sequenzcursor-basierten Verarbeitung *nicht* genutzten) Alternativen denkbar:

- *Alternative 1*

  Kann der Sequenzcursor nicht soweit nach rechts bewegt werden, wie gefordert, wird er zumindest soweit nach rechts bewegt, wie möglich. Der Sequenzcursor würde also auf die Wurzel des rechtesten Baums positioniert.

  Es ist fraglich, ob eine solche „Ausweichspositionierung“ überhaupt erwünscht ist. Die Anwendungslogik könnte ja vorsehen, auf die Positionierung zu verzichten, falls diese nicht wie gewünscht durchführbar ist. Durch die automatische Positionierung auf den rechtesten Wurzelknoten wäre die ursprüngliche Sequenzcursorposition jedoch verloren gegangen und könnte vom Anwendungsprogramm im Allgemeinen auch nicht rekonstruiert werden.

- *Alternative 2*

  Kann der Sequenzcursor nicht soweit nach rechts bewegt werden, wie gefordert, wird er rechts *neben* den letzten Wurzelknoten (also außerhalb der Sequenz) positioniert.

  Dieses Vorgehen hätte keinerlei Vorteile gegenüber der zuvor betrachteten Alternative 1. Insbesondere würde auch hier die ursprüngliche Sequenzcursorposition unwiederbringlich verloren gehen.

- *Alternative 3*

  Kann der Sequenzcursor nicht soweit nach rechts bewegt werden, wie gefordert, wird er in den Status *„unpositioniert“* versetzt (wodurch er seine bisherige Position verliert).

  Auch hier ginge die ursprüngliche Sequenzcursorposition dauerhaft verloren, sodass ein erneuter Positionierungsversuch (mit einer anderen relativen Positionierungsanweisung) nicht möglich wäre. Im Vergleich zu den beiden anderen Alternativen besäße dieses Vorgehen keine Vorteile.

Die zuvor anhand des konkreten Beispiels vorgenommenen Betrachtungen (die sich sinngemäß auf andere Situationen und insbesondere auf andere Positionierungsanweisungen übertragen lassen) zeigen, dass eine Sequenzcursorpositionierung komplett scheitern sollte, falls sie nicht wie gewünscht ausgeführt werden kann. Es ist also nicht sinnvoll, eine „Ausweich-“ bzw. „Ersatzpositionierung“ durchzuführen — der betreffende Sequenzcursor sollte stattdessen seine Position (und seinen Status) beibehalten.

5.4 Möglichkeiten für den Status

Im Abschnitt 4.1 hatten wir zunächst den (nicht-erweiterten) Sequenzcursor-basierten Verarbeitungsablauf eingeführt. Darauf aufbauend haben wir dann im Abschnitt 4.3 den

erweiterten Sequenzcursor-basierten Verarbeitungsablauf betrachtet. Die beiden Varianten des Ablaufs sind in Abbildung 4.1 bzw. 4.26 dargestellt.

Die Möglichkeiten, die es für den Status eines Sequenzcursors gibt, hängen (anders als beispielsweise die Positionierungsmöglichkeiten) davon ab, ob wir den *nicht-erweiterten* oder den *erweiterten* Sequenzcursor-basierten Verarbeitungsablauf zugrunde legen. Nach einer Erläuterung der Statusmöglichkeiten für den Fall des *nicht-erweiterten* Ablaufs im Abschnitt 5.4.1 befassen wir uns im Abschnitt 5.4.2 mit den Statusmöglichkeiten beim *erweiterten* Verarbeitungsablauf.

5.4.1 Statusmöglichkeiten beim nicht-erweiterten Ablauf

Den nicht-erweiterten Sequenzcursor-basierten Verarbeitungsablauf vorausgesetzt, wird nur zwischen den beiden folgenden Statusmöglichkeiten für Sequenzcursor unterschieden:

- *„positioniert“*

  Dieser Status besagt, dass der entsprechende Sequenzcursor auf einem bestimmten Knoten steht. Damit ist es möglich, bei der Definition eines Sequenzausschnitts auf diesen Sequenzcursor Bezug zu nehmen oder den Sequenzcursor mit Hilfe einer relativen Positionierungsanweisung weiterzubewegen.

- *„unpositioniert“*

  Durch diesen Status wird signalisiert, dass der betreffende Sequenzcursor nicht auf einem Knoten positioniert ist. Der Sequenzcursor lässt sich somit weder bei einer Sequenzausschnittsdefinition nutzen, noch kann er mittels einer relativen Positionierungsanweisung bewegt werden.

Wie im Abschnitt 5.1.3 geschildert, befindet sich ein Sequenzcursor nach seiner Erzeugung stets im Status *„unpositioniert“*. Bei einem Weiterbewegen des (dem Sequenzcursor zugeordneten) Tupelcursors bleibt der Status *„unpositioniert“* erhalten — eine Überführung in den Status *„positioniert“* ist ausschließlich durch eine erfolgreiche absolute Positionierung möglich.

Wird ein Sequenzcursor, der sich im Status *„positioniert“* befindet, erfolgreich (relativ oder absolut) positioniert, so ändert sich nichts an seinem Status. Gleiches gilt, wenn eine relative oder absolute Positionierung des Sequenzcursors scheitert (vgl. Abschnitt 5.3.2). Wird der Tupelcursor (der dem Sequenzcursor zugeordnet ist) weiterbewegt, so wird der Sequenzcursor dadurch in den Status *„unpositioniert“* versetzt (vgl. Abschnitt 5.1.3). *Abbildung 5.18* fasst die Statusmöglichkeiten sowie die Statusübergänge nochmals kompakt zusammen.

Beim nicht-erweiterten Sequenzcursor-basierten Verarbeitungsablauf folgt auf das Einbringen der lokalen Änderungen unmittelbar das Weiterbewegen des Tupelcursors, wodurch die (zum entsprechenden Tupelcursor gehörenden) Sequenzcursor in den Status *„unpositioniert“* überführt werden. Sollte das Einbringen der lokalen Änderungen dazu führen, dass ein Knoten gelöscht wird, auf dem ein Sequenzcursor positioniert ist, so ist dies ohne Bewandtnis, da auf die entsprechende Sequenzcursorposition ohnehin kein Bezug mehr genommen wird — der Sequenzcursor wird im nächsten Verarbeitungsschritt ja garantiert in den Status *„unpositioniert“* versetzt. Im Gegensatz zum erweiterten Verarbeitungsablauf

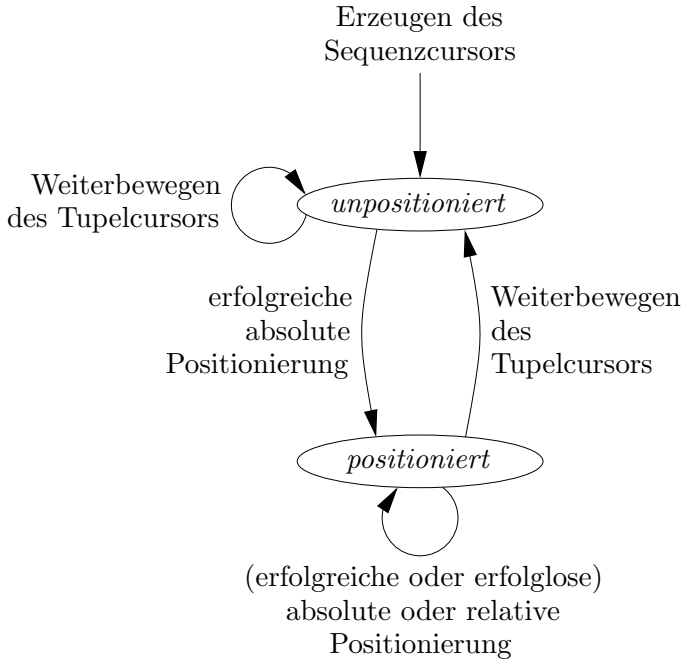


Abbildung 5.18: Statusmöglichkeiten (nicht-erweiterter Ablauf)

(vgl. Abschnitt 5.4.2) sind somit keine Statusmöglichkeiten wie *„zwischen zwei Bäumen“* oder *„unterhalb eines Knotens“* notwendig [Böh07].

5.4.2 Statusmöglichkeiten beim erweiterten Ablauf

Im Rahmen des erweiterten Sequenzcursor-basierten Verarbeitungsablaufs existieren für Sequenzcursor die folgenden sechs Statusmöglichkeiten:

- *„positioniert“*

  Wie beim nicht-erweiterten Ablauf drückt dieser Status aus, dass der Sequenzcursor auf einem bestimmten Knoten steht.

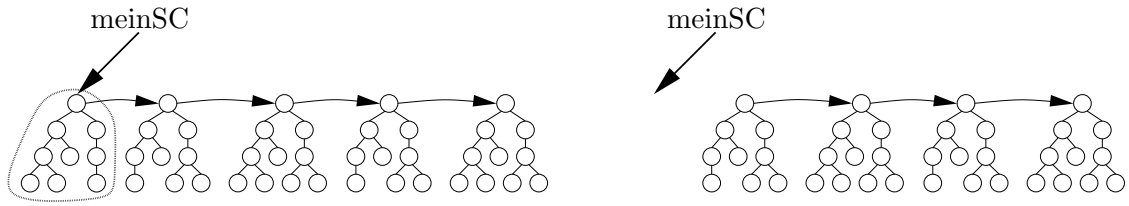
- *„links neben der Sequenz“*

  Dieser Status besagt, dass der Sequenzcursor nicht auf einen Knoten positioniert ist, sondern sich links von der Sequenz (also links vom ersten Baum) befindet. Eine Möglichkeit, in diesen Status zu gelangen, besteht darin, dass der Sequenzcursor (zunächst) auf der Wurzel des linkesten Baums steht und dieser Baum beim Einbringen der lokalen Änderungen gelöscht wird *(Abbildung 5.19).*[10]

- *„rechts neben der Sequenz“*

  Durch diesen Status wird signalisiert, dass der Sequenzcursor rechts von der Sequenz, also rechts vom letzten Baum steht. Dieser Status wird beispielsweise angenommen,

[10] In Abbildung 5.19 (sowie in den folgenden Abbildungen) kennzeichnet die Markierung innerhalb der linken Sequenz denjenigen Baum, der beim Einbringen der lokalen Änderungen gelöscht wird.

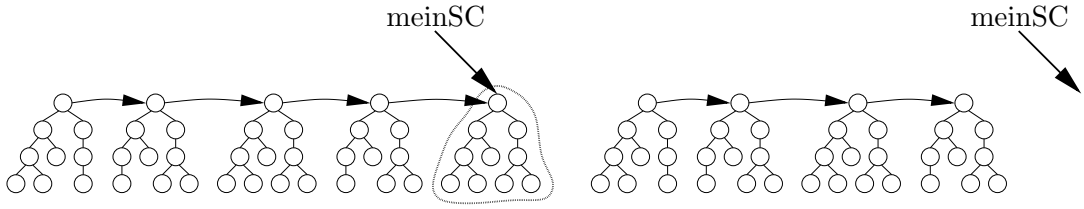


vor dem Einbringen der lokalen Änderungen

nach dem Einbringen der lokalen Änderungen

Abbildung 5.19: Übergang in den Status *„links neben der Sequenz“*

wenn der Sequenzcursor auf die Wurzel des rechtesten Baums positioniert ist und dieser Baum infolge des Einbringens der lokalen Änderungen gelöscht wird *(Abbildung 5.20)*.



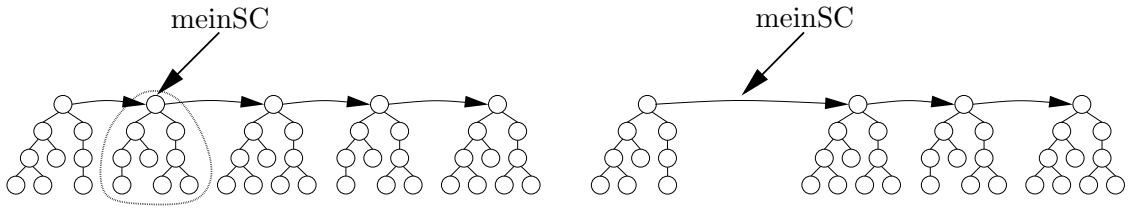
vor dem Einbringen der lokalen Änderungen

nach dem Einbringen der lokalen Änderungen

Abbildung 5.20: Übergang in den Status *„rechts neben der Sequenz“*

- *„zwischen zwei Bäumen“*

  Dieser Status drückt aus, dass der Sequenzcursor zwischen zwei benachbarten Bäumen positioniert ist. Ein Sequenzcursor wird z. B. dann in diesen Status versetzt, wenn er auf der Wurzel eines Baums steht (bei welchem es sich weder um den ersten noch um den letzten Baum handelt) und das Einbringen der lokalen Änderungen zum Löschen dieses Baums führt *(Abbildung 5.21)*.



vor dem Einbringen der lokalen Änderungen

nach dem Einbringen der lokalen Änderungen

Abbildung 5.21: Übergang in den Status *„zwischen zwei Bäumen“*

- *„unterhalb eines Knotens“*

  Dieser Status signalisiert, dass der Sequenzcursor nicht auf einem Knoten steht,

sondern sich unterhalb eines Knotens befindet. Dieser Status wird beispielsweise angenommen, wenn der Sequenzcursor auf einen inneren Baumknoten positioniert ist und dieser Knoten infolge des Einbringens der lokalen Änderungen gelöscht wird *(Abbildung 5.22)*.[11]

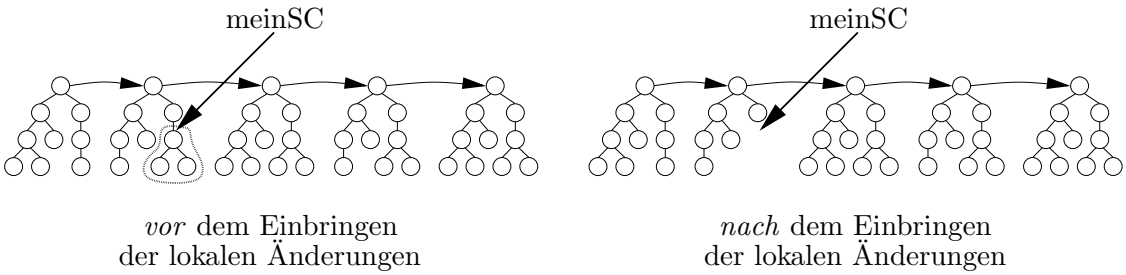


Abbildung 5.22: Übergang in den Status *„unterhalb eines Knotens“*

- *„unpositioniert“*

  Durch diesen Status wird ausgedrückt, dass der Sequenzcursor nicht auf einen Knoten positioniert ist und er sich weder links bzw. rechts neben der Sequenz noch zwischen zwei Bäumen noch unterhalb eines Knotens befindet. In diesen Status wird ein Sequenzcursor z. B. bei seiner Erzeugung oder beim Weiterbewegen des (ihm zugeordneten) Tupelcursors versetzt.

Besitzt ein Sequenzcursor den Status *„links neben der Sequenz“*, *„rechts neben der Sequenz“*, *„zwischen zwei Bäumen“* oder *„unterhalb eines Knotens“*, so kann er durch eine erfolgreiche absolute oder relative Positionierung in den Status *„positioniert“* überführt werden. Eine erfolglose Positionierung würde hingegen nichts am Status ändern. Wird der (zum Sequenzcursor gehörende) Tupelcursor weiterbewegt, so besitzt der Sequenzcursor — unabhängig von seinem aktuellen Status — anschließend den Status *„unpositioniert“*.

Befindet sich ein Sequenzcursor unmittelbar vor dem Einbringen der lokalen Änderungen im Status *„positioniert“* — d. h., der Sequenzcursor steht auf einem bestimmten Knoten — so sind prinzipiell die folgenden drei Fälle möglich:

- Fall 1

  Der Knoten, auf dem der Sequenzcursor positioniert ist, wird durch das Einbringen der lokalen Änderungen *nicht* gelöscht. Somit steht der Sequenzcursor anschließend immer noch auf diesem Knoten — er behält also den Status *„positioniert“* bei.

- Fall 2

  Beim Einbringen der lokalen Änderungen wird der Knoten gelöscht, auf dem der Sequenzcursor steht, wobei jedoch *nicht* alle Knoten der Sequenz gelöscht werden. Der Sequenzcursor befindet sich im Anschluss somit im Status *„links neben der Sequenz“*, *„rechts neben der Sequenz“*, *„zwischen zwei Bäumen“* oder *„unterhalb eines Knotens“*.

[11] Da es nicht möglich ist, dass ein Knoten gelöscht wird, ohne dass auch dessen Kindknoten, Enkelknoten usw. gelöscht werden, existiert weder ein Status *„oberhalb eines Knotens“* noch ein Status *„vertikal zwischen zwei Knoten“*.

- Fall 3

  Durch das Einbringen der lokalen Änderungen werden sämtliche Knoten der Sequenz gelöscht. Der Sequenzcursor besitzt anschließend somit den Status *„unpositioniert“*.

Für die Übergänge zwischen den Statusmöglichkeiten *„unpositioniert“* und *„positioniert“* gelten zudem die Aussagen aus Abschnitt 5.4.1. Alle Statusübergänge sind in *Abbildung 5.23* nochmals überblicksweise dargestellt. Die Bezeichnung *„teilpositioniert“* dient dabei als Sammelbegriff für die Statusmöglichkeiten *„links neben der Sequenz“*, *„rechts neben der Sequenz“*, *„zwischen zwei Bäumen“* und *„unterhalb eines Knotens“*.

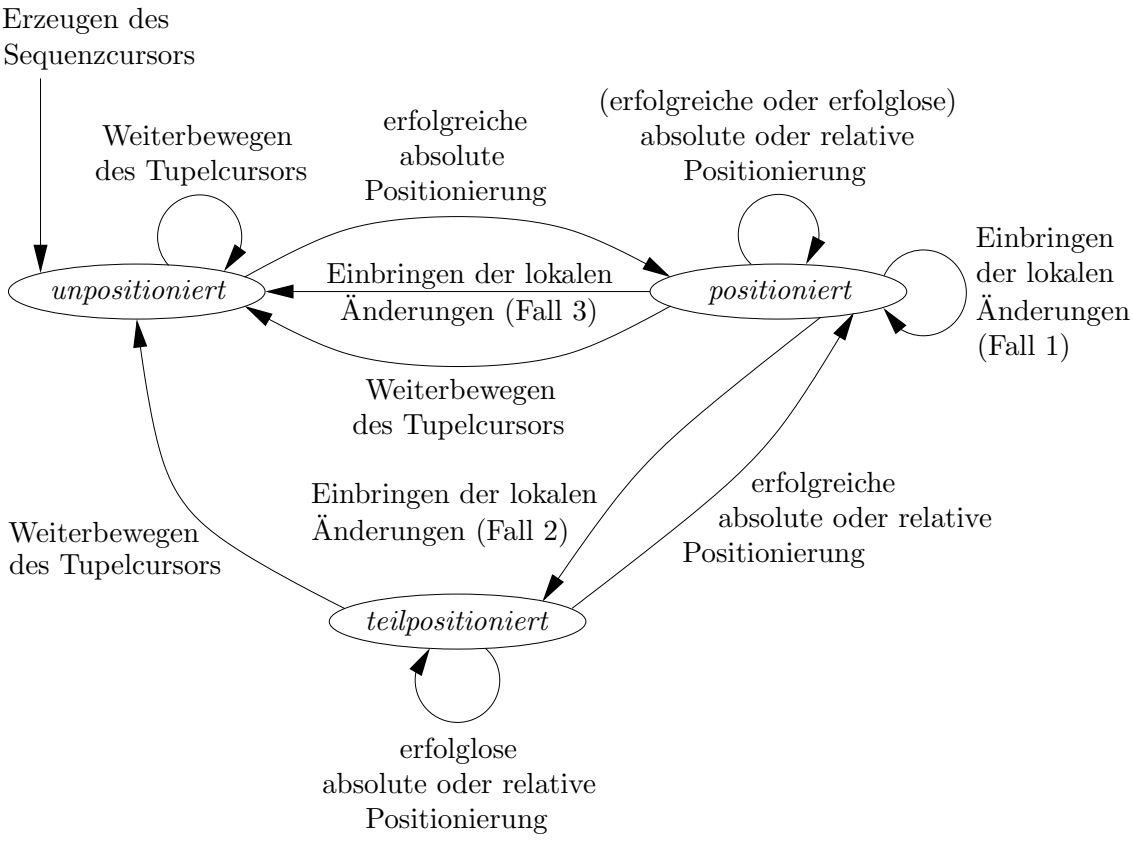


Abbildung 5.23: Statusmöglichkeiten (erweiterter Ablauf)

Kapitel 6

Sequenzausschnitte

Bei der Sequenzcursor-basierten Verarbeitung (Kapitel 4) werden mit Hilfe von Sequenzcursorn Ausschnitte der im aktuellen Ergebnistupel enthaltenen (XQuery-)Sequenzen definiert. Diese Sequenzausschnitte [Mül06, Böh07] werden ins Anwendungsprogramm übertragen und stehen dort für eine lokale Verarbeitung zur Verfügung. Sofern lokal *ändernd* auf den Sequenzausschnitten gearbeitet wird, kann das Anwendungsprogramm entscheiden, ob die vorgenommenen Änderungen in die Datenbank (wieder)eingebracht werden sollen oder nicht.

Im aktuellen Kapitel gehen wir ausführlich auf die Sequenzausschnitte ein. Abschnitt 6.1 beschreibt zunächst ihre grundlegenden Eigenschaften. Eine dieser Eigenschaften besagt, dass Sequenzausschnitte aus mehreren (Sequenzausschnitts-)Teilen bestehen können. Wir werden die Sequenzausschnittsteile und deren Klassifizierung im Abschnitt 6.2 näher betrachten. Abschnitt 6.3 beschäftigt sich anschließend mit dem Definieren von Sequenzausschnitten. Die Übertragung ins Anwendungsprogramm ist dann Gegenstand von Abschnitt 6.4. Im Anschluss widmet sich Abschnitt 6.5 der lokalen Verarbeitung der Sequenzausschnitte. Erläuterungen zum Einbringen der beim lokalen Arbeiten vorgenommenen Änderungen enthält Abschnitt 6.6. Abschließend geht es im Abschnitt 6.7 um die Frage, wie das Konzept der Knotenidentität im Rahmen der Sequenzcursor-basierten Verarbeitung berücksichtigt wird.

6.1 Grundlegende Eigenschaften

In den nun folgenden Abschnitten 6.1.1 bis 6.1.6 werden wir die grundlegenden Eigenschaften der Sequenzausschnitte genauer betrachten.

6.1.1 Implizite und explizite Löschbarkeit

Unter dem Löschen eines Sequenzausschnitts verstehen wir im weiteren Verlauf dieser Arbeit nicht das Löschen der im Sequenzausschnitt enthaltenen Daten, sondern das Löschen der Sequenzausschnittsdefinition. Umfasst ein Sequenzausschnitt beispielsweise die letzten fünf Bäume einer Sequenz, so bedeutet ein Löschen des Sequenzausschnitts *nicht*, dass diese fünf Bäume gelöscht werden — stattdessen wird lediglich die Information gelöscht, dass ein Sequenzausschnitt existiert, der diese fünf Bäume umfasst.

Ein Sequenzausschnitt kann *implizit* oder *explizit* gelöscht werden. Ein *implizites* (und damit automatisches) Löschen findet statt, wenn der Tupelcursor (der zum Anfrageergebnis gehört, das den Sequenzausschnitt enthält) weiterbewegt oder geschlossen wird. Ein solches implizites Löschen ist sinnvoll, da ein Sequenzausschnitt nach dem Weiterbewegen oder Schließen des Tupelcursors nicht länger gebraucht wird. Dies gilt sowohl für den nicht-erweiterten als auch für den erweiterten Sequenzcursor-basierten Verarbeitungsablauf.

Für das *explizite* Löschen eines Sequenzausschnitts steht die Sequenzcursoranweisung `UNDEFINE SEQUENCE PART` zur Verfügung, die im Abschnitt 7.3.2 genauer vorgestellt wird. Das explizite Löschen eines Sequenzausschnitts kann sinnvoll sein, wenn das Anwendungsprogramm (vor dem Weiterbewegen bzw. Schließen des Tupelcursors) feststellt, dass der entsprechende Sequenzausschnitt nicht länger benötigt wird.

6.1.2 Existenz nur im aktuellen Ergebnistupel

Die Definition eines Sequenzausschnitts erfolgt mit Bezugnahme auf einen oder mehrere Sequenzcursor. Da Sequenzcursor stets nur im jeweils aktuellen Ergebnistupel positioniert sein können, befindet sich ein Sequenzausschnitt zum Zeitpunkt seiner Definition garantiert im aktuellen Ergebnistupel — also in dem Ergebnistupel, auf dem der (den Sequenzcursorn zugeordnete) Tupelcursor aktuell steht.

Beim Ergebnistupel, in dem der Sequenzausschnitt enthalten ist, handelt es sich so lange um das *aktuelle* Ergebnistupel, bis der entsprechende Tupelcursor weiterbewegt oder geschlossen wird. Ein Weiterbewegen oder Schließen des Tupelcursors würde aber ein implizites Löschen des Sequenzausschnitts nach sich ziehen (Abschnitt 6.1.1). Der Sequenzausschnitt würde also gelöscht, bevor das Ergebnistupel, das ihn enthält, nicht mehr das aktuelle Ergebnistupel ist. Folglich befindet sich der Sequenzausschnitt während seiner gesamten Lebensdauer im *aktuellen* Ergebnistupel.

6.1.3 Begrenzung auf einen Sequenzausschnitt pro Sequenz

Für jede im Anfrageergebnis enthaltene XQuery-Sequenz darf zu jedem Zeitpunkt jeweils nur ein Sequenzausschnitt definiert sein [Ott07]. Es ist also nicht zulässig, dass eine Sequenz gleichzeitig mehrere Sequenzausschnitte enthält.

In gewissen Fällen scheint es auf den ersten Blick jedoch hilfreich zu sein, wenn — entgegen der obigen Forderung — mehrere Ausschnitte ein und derselben Sequenz definiert werden könnten. Wir möchten dies kurz anhand eines Beispiels betrachten und nehmen an, dass zur lokalen Verarbeitung im Anwendungsprogramm ausschließlich das erste und letzte Bonusangebot — also der erste und der letzte Baum der Bonusangebotesequenz — benötigt werden. Es scheint nun naheliegend, *zwei* Sequenzausschnitte zu definieren, wobei der eine den ersten Baum und der andere den letzten Baum umfasst. Die Auswahl der beiden Bäume lässt sich jedoch auch mit Hilfe eines einzelnen Sequenzausschnitts realisieren. Dies ist möglich, da ein Sequenzausschnitt nicht zusammenhängend zu sein braucht (Abschnitt 6.1.4). Es kann also ein (nicht-zusammenhängender, aus zwei Teilen bestehender) Sequenzausschnitt definiert werden, der ausschließlich den ersten und letzten Baum umfasst.

Wegen der Zulässigkeit nicht-zusammenhängender Sequenzausschnitte führt die Begrenzung auf einen Sequenzausschnitt pro Sequenz also *nicht* zu einer Einschränkung der Flexibilität, mit der Teile einer Sequenz ausgewählt werden können: Statt (auf ein und derselben Sequenz) mehrere separate Sequenzausschnitte zu definieren, wird einfach ein einzelner Sequenzausschnitt festgelegt, der aus mehreren (nicht miteinander verbundenen) Teilen besteht.

6.1.4 Zulässigkeit nicht-zusammenhängender Sequenzausschnitte

Ein Sequenzausschnitt muss (im Sinne der Graphentheorie) nicht zwangsläufig zusammenhängend sein, sondern kann aus mehreren nicht miteinander verbundenen Teilen — den so genannten *Sequenzausschnittsteilen* — bestehen. Der in *Abbildung 6.1* gezeigte Sequenzausschnitt setzt sich beispielsweise aus vier Sequenzausschnittsteilen zusammen. Jeder Sequenzausschnittsteil (kurz *SAT*) ist für sich betrachtet jeweils zusammenhängend.

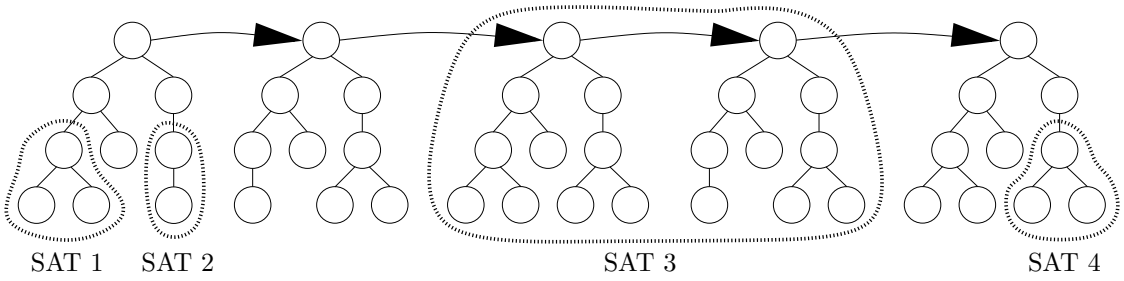


Abbildung 6.1: Sequenzausschnitt, bestehend aus vier Sequenzausschnittsteilen

Die einzelnen Sequenzausschnittsteile eines Sequenzausschnitts sind (implizit) mit 1 beginnend durchnummeriert. Bei der lokalen Verarbeitung eines Sequenzausschnitts kann dann mit Hilfe der (Sequenzausschnittsteile-)Nummern auf die einzelnen Sequenzausschnittsteile Bezug genommen werden. Die Nummerierung der Sequenzausschnittsteile lässt sich wie folgt ermitteln:

- *1. Schritt:*

  Sämtliche Knoten der Sequenz werden (mit 1 beginnend) durchnummeriert, wobei im linkesten Baum der Sequenz begonnen wird. Nachdem allen Knoten des linkesten Baums eine Nummer zugeordnet wurde, wird die Nummerierung im zweitlinkesten Baum fortgesetzt. Anschließend erhalten alle Knoten des drittlinkesten Baums eine Nummer usw. Innerhalb jedes Baums erfolgt die Nummerierung jeweils gemäß der Präorder-Reihenfolge. *Abbildung 6.2* zeigt die Knotennummerierung für unsere Beispielsequenz aus Abbildung 6.1.

- *2. Schritt:*

  Für jeden Sequenzausschnittsteil wird jeweils die kleinste Knotennummer ermittelt. In unserem Beispiel sind dies die Nummern *3*, *8*, *19* und *45* (vgl. Abbildung 6.2).

- *3. Schritt:*

  Die Sequenzausschnittsteile werden entsprechend der Reihenfolge ihrer kleinsten

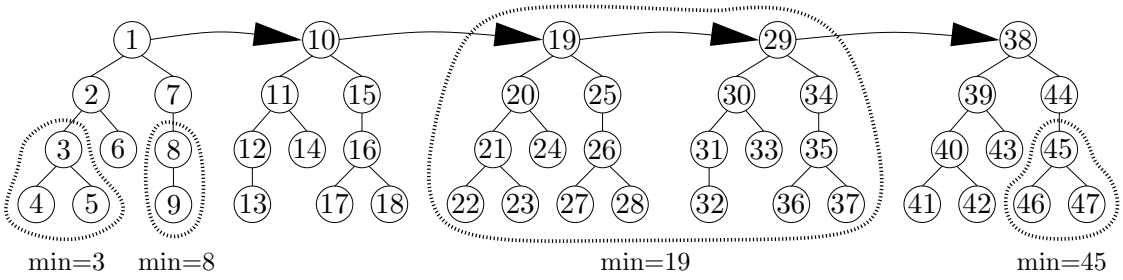


Abbildung 6.2: Sequenz mit nummerierten Knoten

Knotennummern durchnummeriert. Für unser Beispiel ergibt sich damit die in Abbildung 6.1 dargestellte Nummerierung der Sequenzausschnittsteile.

Die Alternative zum (von uns vorgesehenen) Zulassen nicht-zusammenhängender Sequenzausschnitte bestände darin, zu fordern, dass Sequenzausschnitte stets zusammenhängend sein müssen. In Kombination mit der Festlegung, dass pro Sequenz nur ein Sequenzausschnitt erlaubt ist (Abschnitt 6.1.3), würde dies jedoch zu starken Einschränkungen bei der Auswahl der ins Anwendungsprogramm zu übertragenden Sequenzteile führen. So wäre es beispielsweise nicht möglich, ausschließlich den ersten und den siebenten Baum einer (Bonusangebote-)Sequenz auszuwählen — zumindest die Wurzelknoten der fünf dazwischenliegenden Bäume müssten zwangsläufig mit ins Anwendungsprogramm übertragen werden.

6.1.5 Namenlosigkeit

Einem Sequenzausschnitt wird beim Definieren kein Name zugeordnet — Sequenzausschnitte sind also namenlos. Auf eine Benennung/Benamung der Sequenzausschnitte wird verzichtet [Böh07], da es auch ohne Namen möglich ist, gezielt auf einen bestimmten Sequenzausschnitt Bezug zu nehmen. Dies gilt sowohl für eine Bezugnahme innerhalb von Sequenzcursoranweisungen als auch für eine Bezugnahme beim lokalen Arbeiten. Wir werden hierauf im Folgenden näher eingehen.

- *Bezugnahme innerhalb von Sequenzcursoranweisungen*

  Eindeutige Ergebnisspaltennamen vorausgesetzt, kann ein Sequenzausschnitt durch die Kombination aus Ergebnisspaltenname und Tupelcursorname identifiziert werden. Der Tupelcursorname (z. B. *unserTC*) dient dabei zur Bezugnahme auf ein bestimmtes Anfrageergebnis.[1] Durch die Angabe des Spaltennamens (z. B. *C*) wird spezifiziert, in welcher Spalte dieses Anfrageergebnisses der Sequenzausschnitt enthalten ist. Da ein Sequenzausschnitt nur im aktuellen Ergebnistupel existieren kann (Abschnitt 6.1.2), steht auch fest, welche Zeile des Anfrageergebnisses den Sequenzausschnitt enthält — nämlich die Zeile, auf der der entsprechende Tupelcursor positioniert ist. Somit ist eindeutig bestimmt, zu welchem Spaltenwert — d. h. zu welcher Sequenz — der Sequenzausschnitt gehört (vgl. *Abbildung 6.3*). Da in dieser Sequenz

[1] Es können zugleich mehrere Anfrageergebnisse existieren, wobei jedem Anfrageergebnis jeweils ein anderer Tupelcursor zugeordnet ist.

nur ein einziger Sequenzausschnitt existieren kann (Abschnitt 6.1.3), ist damit auch der entsprechende Sequenzausschnitt eindeutig identifiziert.

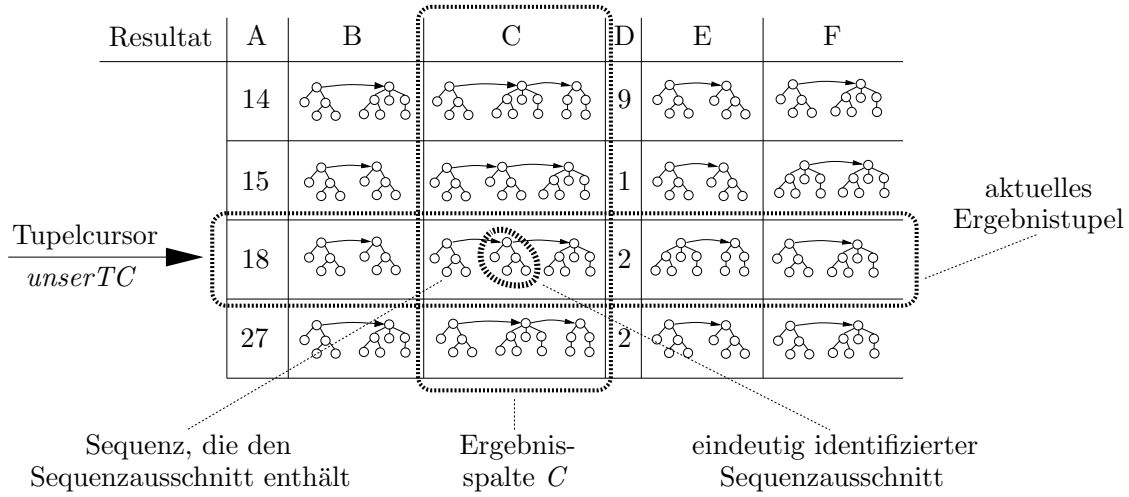


Abbildung 6.3: Bezugnahme auf einen Sequenzausschnitt

Bei der Bezugnahme auf einen Sequenzausschnitt kann bei Bedarf (z. B. wenn mehrere gleichnamige Ergebnisspalten existieren) anstelle des Ergebnisspaltennamens auch die Ergebnisspaltennummer verwendet werden. Die Identifikation des Sequenzausschnitts erfolgt dann mittels der Kombination aus Ergebnisspaltennummer und Tupelcursorname.

- *Bezugnahme beim lokalen Arbeiten*

  Lokal im Anwendungsprogramm kann (beispielsweise wenn das Anfrageergebnis mehrere XML-Spalten besitzt) gleichzeitig auf mehreren Sequenzausschnitten gearbeitet werden. Die Bezugnahme auf einen bestimmten Sequenzausschnitt erfolgt dabei mit Hilfe einer Zeigervariable, die auf das Objekt zeigt, welches für die lokale Verwaltung des entsprechenden Sequenzausschnitts verantwortlich ist (vgl. Abschnitt 4.2.6 bzw. 4.2.7).

6.1.6 Typed-value-orientierte Repräsentation als Basis

Wie bereits aus Abschnitt 4.1 bekannt, basieren die Sequenzausschnitte auf der im Kapitel 3 vorgestellten typed-value-orientierten Repräsentation von Sequenzen: Sowohl das Definieren der Sequenzausschnitte als auch das lokale Verarbeiten der Sequenzausschnitte erfolgt auf der Grundlage dieser von uns konzipierten Repräsentationsvariante. Eine Konsequenz hiervon ist beispielsweise, dass ein Sequenzausschnitt zwar V-Knoten, aber keine N-Knoten enthalten kann.

Warum die typed-value-orientierte Repräsentation von Sequenzen (und nicht die traditionelle Sequenzrepräsentation) verwendet wird, haben wir ausführlich im Abschnitt 3.1 motiviert.

6.2 Sequenzausschnittsteile und deren Klassifizierung

Ein Sequenzausschnitt besteht stets aus einem oder mehreren (für sich betrachtet, jeweils zusammenhängenden) Sequenzausschnittsteilen [Böh07]. Wir werden im aktuellen Abschnitt aufzeigen, welche Möglichkeiten es für die Gestalt eines Sequenzausschnittsteils gibt. Hierzu führen wir eine Klassifizierung der Sequenzausschnittsteile durch, die auf den folgenden drei Kriterien basiert:

(1) Der Sequenzausschnittsteil enthält Knoten aus *mehreren* Bäumen der Sequenz.

(2) Der Sequenzausschnittsteil enthält *keinen* Knoten aus der Wurzelebene der Sequenz.

(3) Bei der Festlegung des Sequenzausschnittsteils wurden bestimmte (Unter)-Teilbäume ausgeschlossen, die somit nicht zum Sequenzausschnittsteil gehören.

Für einen konkreten Sequenzausschnittsteil ist jedes der drei zuvor genannten Kriterien entweder erfüllt oder nicht erfüllt. Die sich damit grundsätzlich ergebenden $2^3 = 8$ Kombinationsmöglichkeiten sind in *Abbildung 6.4* aufgeführt.

(1) SAT enthält Knoten aus *mehreren* Bäumen	(2) SAT enthält *keinen* Knoten aus der Wurzelebene	(3) (Unter)-Teilbäume wurden ausgeschlossen	SAT-Klasse
–	–	–	1
–	–	✓	2
–	✓	–	3
–	✓	✓	4
✓	–	–	5
✓	–	✓	6
✓	✓	–	(nicht möglich)
✓	✓	✓	(nicht möglich)

✓ = Kriterium wird erfüllt, – = Kriterium wird nicht erfüllt

Abbildung 6.4: Klassifizierung der Sequenzausschnittsteile

Enthält ein Sequenzausschnittsteil Knoten aus mehreren Bäumen der Sequenz, so muss er zwangsläufig auch Knoten aus der Wurzelebene der Sequenz enthalten — andernfalls könnte der Sequenzausschnittsteil nicht (wie gefordert) zusammenhängend sein. Die Kriterien (1) und (2) können somit nicht gleichzeitig erfüllt werden. Dies hat zur Folge, dass die beiden letzten der in Abbildung 6.4 aufgelisteten Kombinationsmöglichkeiten wegen Unerfüllbarkeit entfallen. Unsere Klassifizierung führt also nur zu sechs (und nicht zu acht) tatsächlich möglichen Klassen. Diese Klassen werden — wie der Abbildung 6.4 entnommen werden kann — als *SAT-Klasse 1*, *SAT-Klasse 2*, ... bzw. *SAT-Klasse 6* bezeichnet.

Wir werden im Folgenden jede der sechs SAT-Klassen näher charakterisieren. Dabei werden wir jeweils zeigen, dass es sinnvoll ist, die Existenz von Sequenzausschnittsteilen der entsprechenden SAT-Klasse zu erlauben.

- *SAT-Klasse 1*

  Bei einem Sequenzausschnittsteil dieser SAT-Klasse handelt es sich zwangsläufig um einen einzelnen kompletten Baum der Sequenz *(Abbildung 6.5)*. Ein derartiger Sequenzausschnittsteil wird beispielsweise benötigt, wenn ausschließlich ein bestimmter Baum (also z. B. ein bestimmtes Bonusangebot) ins Anwendungsprogramm übertragen werden soll.

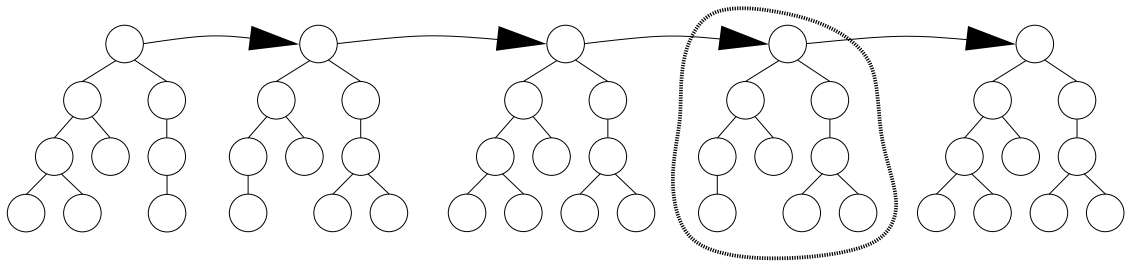


Abbildung 6.5: Beispiel für einen Sequenzausschnittsteil der SAT-Klasse 1

- *SAT-Klasse 2*

  Ein Sequenzausschnittsteil dieser SAT-Klasse ist stets ein einzelner Baum der Sequenz, bei dem ein oder mehrere Teilbäume ausgeschlossen wurden *(Abbildung 6.6)*. Ein solcher Sequenzausschnittsteil wird z. B. verwendet, wenn zur lokalen Verarbeitung nur ein bestimmtes Bonusangebot gebraucht wird und dabei beispielsweise das Sachangebot nicht von Interesse ist.[2]

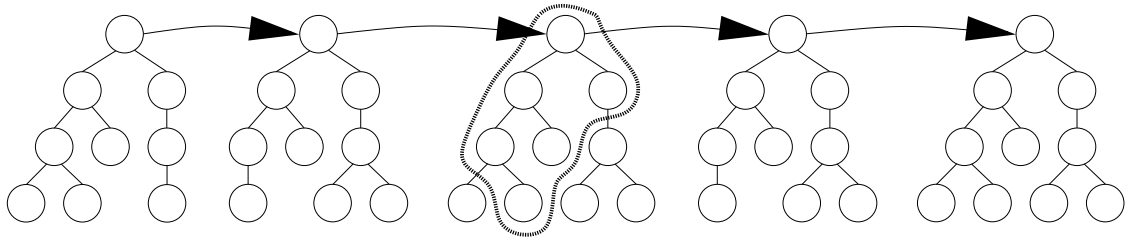


Abbildung 6.6: Beispiel für einen Sequenzausschnittsteil der SAT-Klasse 2

- *SAT-Klasse 3*

  Bei einem Sequenzausschnittsteil, der zu dieser SAT-Klasse gehört, handelt es sich zwangsläufig um einen einzelnen kompletten Teilbaum — also um einen Teilbaum, bei dem keine „Unter-Teilbäume“ ausgeschlossen wurden *(Abbildung 6.7)*. Ein entsprechender Sequenzausschnittsteil wird beispielsweise genutzt, wenn lokal ausschließlich das Sachangebot eines bestimmten Bonusangebots benötigt wird.

- *SAT-Klasse 4*

  Bei einem Sequenzausschnittsteil, der dieser SAT-Klasse zugeordnet ist, handelt es

[2] Wie in Abbildung 3.29 dargestellt, wird das Sachangebot durch einen eigenen Teilbaum repräsentiert. Beim hier angenommenen Sequenzausschnittsteil wurde dieser Teilbaum ausgeschlossen.

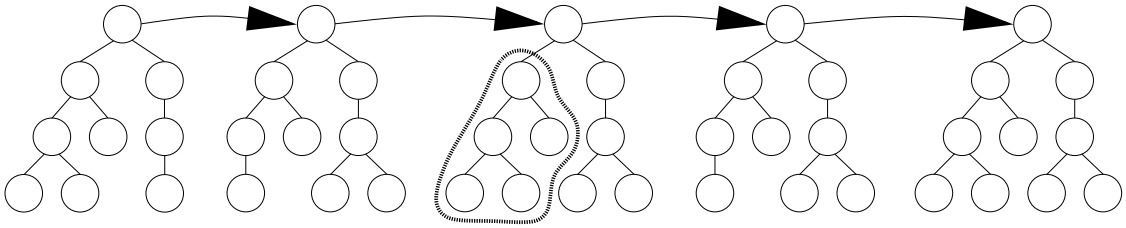


Abbildung 6.7: Beispiel für einen Sequenzausschnittsteil der SAT-Klasse 3

sich stets um einen einzelnen Teilbaum, bei dem ein oder mehrere Unter-Teilbäume ausgeschlossen wurden *(Abbildung 6.8)*. Solch ein Sequenzausschnittsteil wird z. B. benötigt, wenn lokal lediglich das Sachangebot eines bestimmten Bonusangebots verarbeitet werden soll und dabei die Artikelnummer nicht von Interesse ist (vgl. Abbildung 3.29).

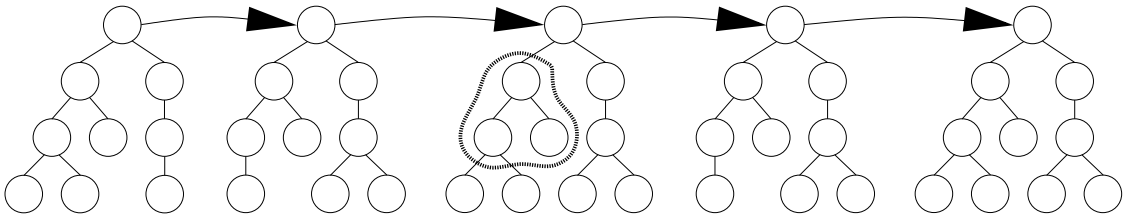


Abbildung 6.8: Beispiel für einen Sequenzausschnittsteil der SAT-Klasse 4

- *SAT-Klasse 5*

  Ein Sequenzausschnittsteil dieser SAT-Klasse besteht zwangsläufig aus mehreren direkt aufeinanderfolgenden kompletten Bäumen der Sequenz *(Abbildung 6.9)*. Im Rahmen unseres Beispielszenarios haben wir einen derartigen Sequenzausschnittsteil genutzt, um die aufwertbaren Bonusangebote ins Anwendungsprogramm zu übertragen (Abschnitt 4.2.5).

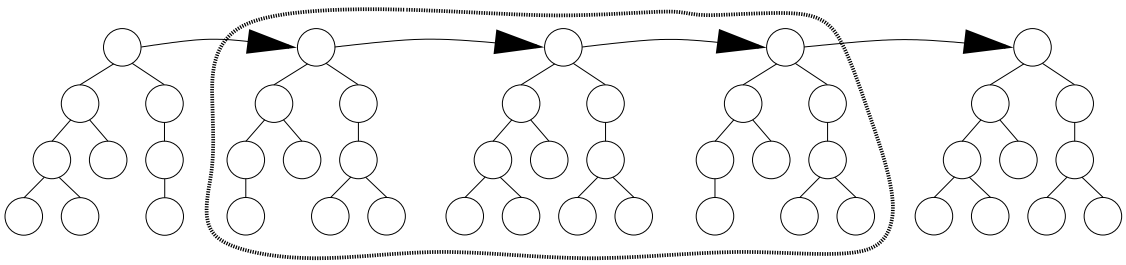


Abbildung 6.9: Beispiel für einen Sequenzausschnittsteil der SAT-Klasse 5

- *SAT-Klasse 6*

  Ein Sequenzausschnittsteil, der zu dieser SAT-Klasse gehört, umfasst stets mehrere direkt aufeinanderfolgende Bäume der Sequenz, bei denen einzelne Teilbäume ausgeschlossen wurden *(Abbildung 6.10)*. Ein entsprechender Sequenzausschnittsteil wird

z. B. verwendet, falls im Anwendungsprogramm mehrere Bonusangebote (beispielsweise die fünf letzten Bonusangebote) gebraucht werden und dabei die Sachangebote nicht von Interesse sind.

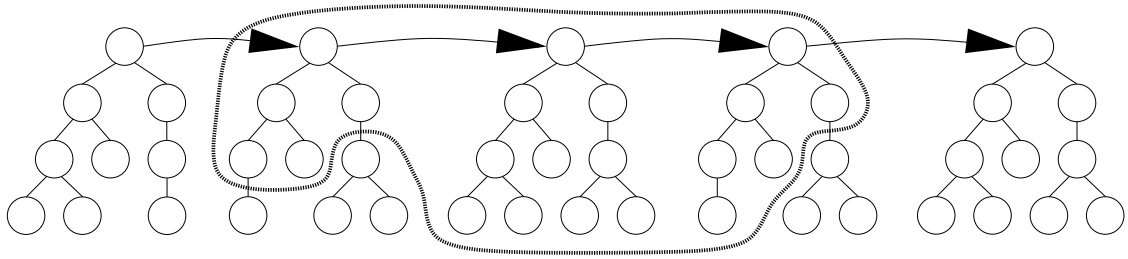


Abbildung 6.10: Beispiel für einen Sequenzausschnittsteil der SAT-Klasse 6

Die von uns vorgenommene Klassifizierung der Sequenzausschnittsteile ist vollständig und disjunkt. Dies bedeutet, dass jeder mögliche bzw. zulässige Sequenzausschnittsteil jeweils zu genau einer SAT-Klasse gehört.

6.3 Definieren von Sequenzausschnitten

Im Abschnitt 6.3.1 betrachten wir zunächst, wie ein *zusammenhängender* Sequenzausschnitt definiert werden kann. Anschließend gehen wir im Abschnitt 6.3.2 auf die Definition *nicht-zusammenhängender* Sequenzausschnitte ein.

Beim Definieren eines (zusammenhängenden oder nicht-zusammenhängenden) Sequenzausschnitts ist es möglich, ein XML-Schema zu spezifizieren, welches dann (u. a.) bei der lokalen Validitätsüberwachung (Abschnitt 6.5.2) nutzbar ist. Für Syntaxdetails diesbezüglich verweisen wir auf Abschnitt 7.3.1, in welchem die zur Definition von Sequenzausschnitten zu verwendende Sequenzcursoranweisung `DEFINE SEQUENCE PART` vorgestellt wird.

6.3.1 Zusammenhängende Sequenzausschnitte

Ein zusammenhängender Sequenzausschnitt besteht aus einem einzigen Sequenzausschnittsteil. Bezüglich dieses Sequenzausschnittsteils lassen sich die beiden folgenden Fälle unterscheiden:

- *Fall 1*

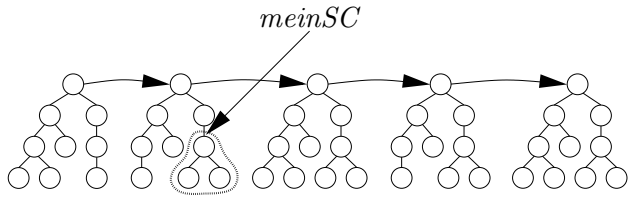
  Der Sequenzausschnittsteil gehört zur SAT-Klasse *1*, *2*, *3* oder *4* und ist somit — im Sinne der Graphentheorie — ein Baum[3].

[3]Dieser Baum muss *nicht* zwangsläufig ein *kompletter* Baum der Sequenz sein. Es kann sich beispielsweise auch um einen *Teil*baum handeln, bei dem bestimmte Unter-Teilbäume ausgeschlossen sind (Abschnitt 6.2).

- *Fall 2*

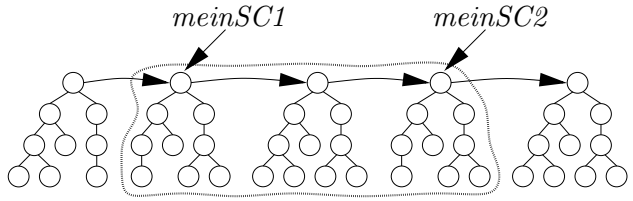
  Der Sequenzausschnittsteil ist der SAT-Klasse *5* oder *6* zuzuordnen und umfasst folglich mehrere direkt aufeinanderfolgende Bäume der Sequenz, bei denen gewisse Teilbäume ausgeschlossen sein können.

Handelt es sich beim Sequenzausschnittsteil (aus dem der zu definierende Sequenzausschnitt bestehen soll) um einen einzelnen Baum *(Fall 1)*, so wird ein Sequenzcursor auf den Wurzelknoten dieses Baums positioniert [Ott07]. Beim Definieren des Sequenzausschnitts wird dann auf diesen Sequenzcursor Bezug genommen *(Abbildung 6.11)*.[4] Besteht der Sequenzausschnittsteil hingegen aus mehreren direkt aufeinanderfolgenden Bäumen der Sequenz *(Fall 2)*, so werden zwei Sequenzcursor genutzt, um die linke bzw. rechte Begrenzung des Sequenzausschnittsteils festzulegen. Die beiden Sequenzcursor werden dabei auf den Wurzelknoten des ersten bzw. letzten zum Sequenzausschnittsteil gehörenden Baums positioniert *(Abbildung 6.12)*.




```
DEFINE SEQUENCE PART AS
meinSC;
```

Abbildung 6.11: Beispiel für die Definition eines Sequenzausschnitts — Fall 1



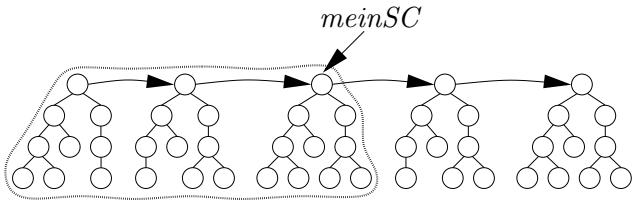

```
DEFINE SEQUENCE PART AS
FROM meinSC1 TO meinSC2;
```

Abbildung 6.12: Beispiel für die Definition eines Sequenzausschnitts — Fall 2

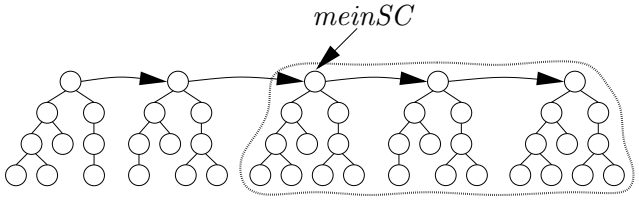
In gewissen Situationen ist es prinzipiell denkbar, die Festlegung eines Sequenzausschnitts bzw. Sequenzausschnittsteils mit weniger Sequenzcursorn als zuvor beschrieben durchzuführen. Dies ist beispielsweise der Fall, wenn ein aus mehreren Bäumen bestehender Sequenzausschnittsteil am Anfang der Sequenz beginnt (also u. a. den linkesten Baum der Sequenz enthält) oder sich der Sequenzausschnittsteil bis zum Ende der Sequenz erstreckt *(Abbildung 6.13)*. Um die Anzahl der syntaktischen Möglichkeiten der Sequenzcursoranweisung `DEFINE SEQUENCE PART` zu beschränken, werden wir im Rahmen dieser Arbeit jedoch auf die Nutzung entsprechender „abgekürzter“ Varianten zur Sequenzausschnittsdefinition verzichten.

Gehört der Sequenzausschnittsteil (aus dem der zu definierende Sequenzausschnitt bestehen soll) zu einer der SAT-Klassen *2*, *4* oder *6*, so werden bei der Definition des Sequenzausschnitts bestimmte (Unter-)Teilbäume ausgeschlossen. Dies geschieht, indem

[4] Nähere Informationen zur Syntax der Sequenzcursoranweisung `DEFINE SEQUENCE PART` finden sich im Abschnitt 7.3.1.



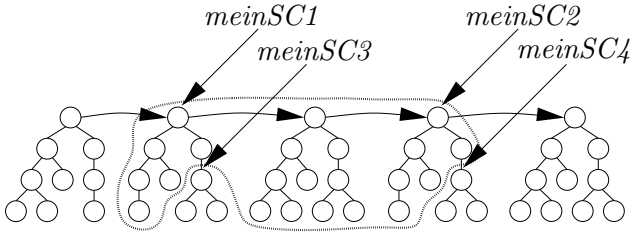

```
DEFINE SEQUENCE PART AS
FROM LEFT BOUNDARY TO meinSC;
```




```
DEFINE SEQUENCE PART AS
FROM meinSC TO RIGHT BOUNDARY;
```

Abbildung 6.13: Beispiele für denkbare „abgekürzte“ Definitionsvarianten

beim Definieren des Sequenzausschnitts auf Sequenzcursor Bezug genommen wird, die zuvor auf die Wurzelknoten der auszuschließenden (Unter-)Teilbäume positioniert wurden *(Abbildung 6.14)*.




```
DEFINE SEQUENCE PART AS
FROM meinSC1 TO meinSC2
WITHOUT meinSC3, meinSC4;
```

Abbildung 6.14: Beispiel für das Ausschließen von Teilbäumen

Beim Ausschließen eines (Unter-)Teilbaums kann Einfluss auf das Verhalten beim späteren Einbringen der lokalen Änderungen genommen werden: Es lässt sich festlegen, was passieren soll, wenn infolge des Einbringens der lokalen Änderungen (unzulässigerweise) ein „freischwebender“ (Unter-)Teilbaum entstehen würde. Dies ist dann der Fall, wenn das Einbringen der lokalen Änderungen dazu führen würde, dass der Vaterknoten vom Wurzelknoten eines beim Definieren des Sequenzausschnitts ausgeschlossenen (Unter-)Teilbaums gelöscht wird *(Abbildung 6.15)*. Entsprechend unserem Sprachvorschlag kann beim Ausschließen eines (Unter-)Teilbaums eine der beiden folgenden Optionen[5] spezifiziert werden:

- `ON DELETE RESTRICT`

  Das Einbringen der lokalen Änderungen scheitert, falls es zum Entstehen eines „freischwebenden“ (Unter-)Teilbaums führen würde.

- `ON DELETE FORCE`

  Würde das Einbringen der lokalen Änderungen zu einem „freischwebenden“ (Unter-)Teilbaum führen, so wird dieser automatisch gelöscht.

[5] Diese Optionen sind an die aus SQL bekannten *referentiellen Aktionen (referential actions)* angelehnt, welche beim Definieren von Fremdschlüsseln festlegbar sind.

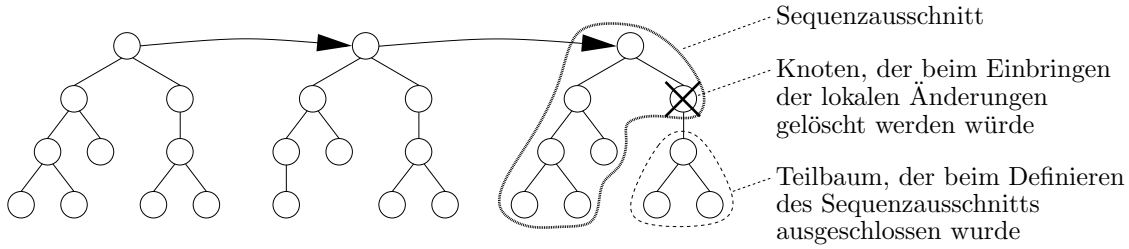


Abbildung 6.15: Zu verhindernde Entstehung eines „freischwebenden“ Teilbaums

Mehrere auszuschließende Teilbäume vorausgesetzt, können innerhalb ein und derselben Sequenzausschnittsdefinition gleichzeitig beide der zuvor genannten Optionen verwendet werden *(Abbildung 6.16)*. Wird keine der beiden Optionen angegeben, wird implizit die Variante `ON DELETE RESTRICT` genutzt.

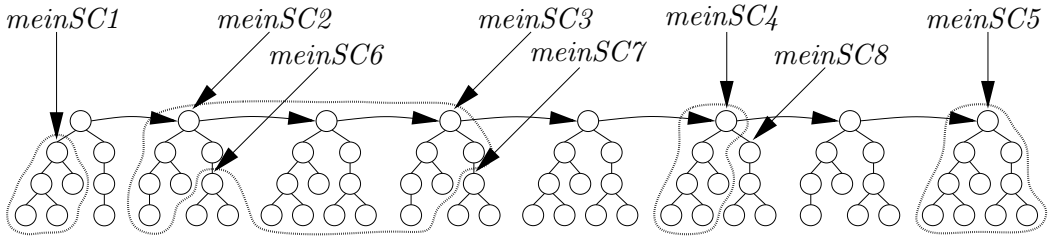
```
DEFINE SEQUENCE PART AS
FROM meinSC1 TO meinSC2
WITHOUT meinSC3 ON DELETE RESTRICT, meinSC4 ON DELETE FORCE;
```

Abbildung 6.16: Beispiel für die gleichzeitige Nutzung beider Optionen

6.3.2 Nicht-zusammenhängende Sequenzausschnitte

Im Abschnitt 6.3.1 haben wir geschildert, wie ein Sequenzausschnitt definiert werden kann, der aus genau einem Sequenzausschnittsteil besteht. Eine solche Definition setzte sich stets aus der einleitenden Schlüsselwortfolge `DEFINE SEQUENCE PART AS` und der Festlegung eines Sequenzausschnittsteils (z. B. `FROM meinSC1 TO meinSC2`) zusammen.

Beim Festlegen eines nicht-zusammenhängenden Sequenzausschnitts — also eines Sequenzausschnitts, der aus *mehreren* Sequenzausschnittsteilen besteht — werden hintereinander einfach *mehrere* Sequenzausschnittsteil-Definitionen angegeben *(Abbildung 6.17)*. Die Definitionen der einzelnen Sequenzausschnittsteile werden dabei mit Hilfe des Schlüsselworts `PLUS` voneinander getrennt [Böh07].




```
DEFINE SEQUENCE PART AS
meinSC1 PLUS
FROM meinSC2 TO meinSC3 WITHOUT meinSC6, meinSC7 PLUS
meinSC4 WITHOUT meinSC8 PLUS
meinSC5;
```

Abbildung 6.17: Beispiel für die Definition eines Sequenzausschnitts

6.4 Übertragung ins Anwendungsprogramm

Wir haben im Abschnitt 4.2.6 beschrieben, welche vorbereitenden Schritte durchzuführen sind, um einen zuvor definierten Sequenzausschnitt ins Anwendungsprogramm zu übertragen. Die eigentliche Übertragung des Sequenzausschnitts wurde dann mit Hilfe der Sequenzcursoranweisung `TRANSFER SEQUENCE PART` veranlasst.

Das Übertragen von Sequenzausschnitten führt dazu, dass im Anwendungsprogramm anschließend lokale Kopien der Sequenzausschnitte verfügbar sind, auf denen mit Hilfe von Sequenzausschnittsmethoden gearbeitet werden kann. Im Abschnitt 6.4.1 gehen wir näher auf die Programmiersprachenrepräsentation dieser Kopien — also auf die lokale Repräsentation der Sequenzausschnitte — ein. Die (eigentliche) Übertragung der Sequenzausschnitte ist dann Gegenstand von Abschnitt 6.4.2.

6.4.1 Lokale Repräsentation der Sequenzausschnitte

Das lokale Arbeiten auf den Sequenzausschnitten erfolgt *ausschließlich* mit Hilfe von Sequenzausschnittsmethoden. Die lokale (Programmiersprachen-)Repräsentation der Sequenzausschnitte ist für den Anwendungsprogrammierer damit vollkommen transparent. Aus konzeptueller Sicht (also insbesondere unter Vernachlässigung etwaiger Auswirkungen auf die Performance) ist es somit völlig unerheblich, welche Repräsentationvariante genutzt wird.

Der Sequenzcursor-basierte Verarbeitungsablauf beruht *nicht* auf einer konkreten Programmiersprachenrepräsentation der Sequenzausschnitte — das Sequenzcursor-basierte Verarbeitungsmodell schreibt also *nicht* vor, in welcher genauen Art und Weise die Sequenzausschnitte lokal zu repräsentieren sind. Die Wahl der Programmiersprachenrepräsentation bleibt stattdessen dem jeweiligen DBMS-Hersteller überlassen, der beabsichtigt, die Sequenzcursor-basierte Verarbeitung in seine Produkte zu integrieren.

Um eine hohe Effizienz sowie eine einfache Implementierbarkeit der Sequenzausschnittsmethoden[6] zu gewährleisten, bietet es sich an, auf eine Objekt- und Zeiger-basierte Repräsentationsvariante zurückzugreifen [Ott07]: Jeder Knoten des Sequenzausschnitts wird dabei als separates Objekt repräsentiert, wobei die Verweise zwischen den Objekten mit Hilfe von Zeigern realisiert werden *(Abbildung 6.18)*.

Alternativ zur Verwendung „verzeigerter Objekte“ ist prinzipiell auch die Nutzung einer anderen (beispielsweise Array-basierten) Repräsentationsvariante denkbar. Für eine Beschreibung und Bewertung der verschiedenen Ansätze verweisen wir auf [Ott07].

6.4.2 Ansätze für die Übertragung

Die Art und Weise der Sequenzausschnittsübertragung ist (wie die lokale Programmiersprachenrepräsentation) für den Anwendungsprogrammierer transparent. Damit ist es aus konzeptueller Sicht irrelevant, wie die Übertragung im Detail durchgeführt wird. Das Sequenzcursor-basierte Verarbeitungsmodell setzt deshalb keine spezielle Übertragungsvariante voraus.

[6] Die Implementierung der Sequenzausschnittsmethoden ist natürlich *nicht* die Aufgabe des Anwendungsprogrammierers: Dem Anwendungsprogrammierer werden verschiedene komfortable Sequenzausschnittsmethoden zur Verfügung gestellt [Ott07].

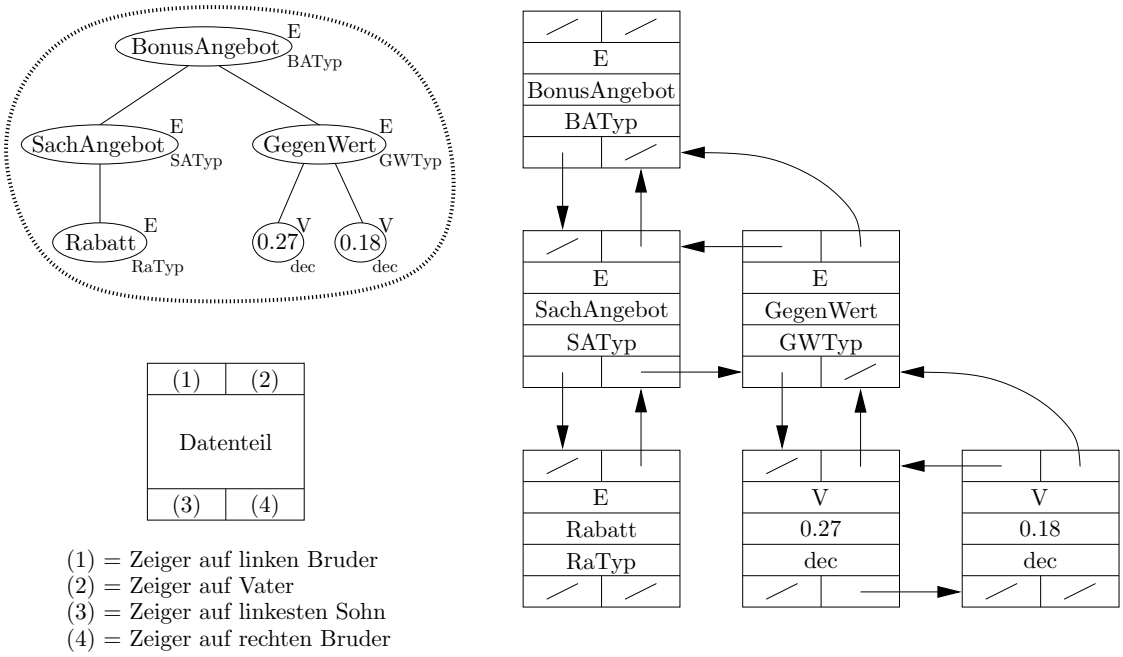


Abbildung 6.18: Sequenzausschnitt und seine (vereinfachte) lokale Repräsentation

Aus Performance- bzw. Aufwandsgründen ist es sinnvoll, ein Vorgehen zu wählen, das es gestattet, den gesamten Sequenzausschnitt in einem einzigen Schritt ins Anwendungsprogramm zu übertragen [Ott07]. Bei einem solchen Vorgehen wird die gewünschte lokale Hauptspeicherrepräsentation des Sequenzausschnitts zunächst komplett vom SQL-Laufzeitsystem „vorberechnet“ und dann in den (zur Aufnahme des Sequenzausschnitts vorgesehenen) lokalen Speicherbereich kopiert.

Bei einer Objekt- und Zeiger-basierten Repräsentationsvariante (Abschnitt 6.4.1) hängen die in den Objekten enthaltenen Zeigerwerte von der (Anfangs-)Adresse des lokal bereitgestellten Speicherbereichs ab, der die Repräsentation des Sequenzausschnitts aufnehmen soll. Dies muss bei der vom SQL-Laufzeitsystem durchzuführenden Berechnung der Hauptspeicherrepräsentation des Sequenzausschnitts berücksichtigt werden: Das SQL-Laufzeitsystem nutzt deshalb die aus der SQLSPCA ausgelesene Adresse des lokalen Speicherbereichs, um (mit Hilfe von „Zeigerarithmetik“) die lokal zu verwendenden Zeigerwerte zu ermitteln. Für eine ausführliche Beschreibung und Bewertung dieses Vorgehens sowie für eine Betrachtung und Bewertung alternativer Ansätze verweisen wir auf [Ott07].

6.5 Lokale Verarbeitung

Im Abschnitt 6.5.1 geht es zunächst um Ansätze für ein lokales Arbeiten auf Sequenzausschnitten. Abschnitt 6.5.2 widmet sich dann der lokalen Validitätsüberwachung. Die Protokollierung von lokal durchgeführten Änderungen ist anschließend Gegenstand von Abschnitt 6.5.3.

6.5.1 Verarbeitungsansätze

Gemäß dem Sequenzcursor-basierten Verarbeitungsmodell erfolgt das lokale Arbeiten auf den Sequenzausschnitten ausschließlich mit Hilfe so genannter Sequenzausschnittsmethoden. Um welche Sequenzausschnittsmethoden es sich dabei genau handelt, wird vom Sequenzcursor-basierten Verarbeitungsmodell jedoch nicht vorgegeben. Für die lokale Verarbeitung der Sequenzausschnitte sind damit prinzipiell verschiedene Ansätze denkbar. Zwei dieser Ansätze werden wir in den Abschnitten 6.5.1.1 und 6.5.1.2 kurz vorstellen.

Bei der Konzeption des Sequenzcursor-basierten Verarbeitungsmodells haben wir bewusst auf eine „starre“ Festlegung von bereitzustellenden Sequenzausschnittsmethoden verzichtet: Die Sequenzausschnittsmethoden sind — im Gegensatz zu den im Rahmen dieser Arbeit eingeführten Sequenzcursoranweisungen — *kein* Bestandteil der von uns vorgeschlagenen SQL-Erweiterung. Selbst eine um Sequenzcursor-basierte Verarbeitungsfunktionalität erweiterte SQL-Norm enthielte also keine Vorgaben darüber, welche Sequenzausschnittsmethoden anzubieten sind. Somit braucht auch das Sequenzcursor-basierte Verarbeitungsmodell keine entsprechenden Vorgaben zu machen. Jede (denkbare) vom Sequenzcursor-basierten Verarbeitungsmodell vorgenommene Festlegung konkreter Sequenzausschnittsmethoden würde folglich eine unnötige Einschränkung (der für ein lokales Arbeiten grundsätzlich nutzbaren Möglichkeiten) darstellen.

6.5.1.1 DOM-artige lokale Verarbeitung

Sequenzausschnittsmethoden, die an die (aus dem Abschnitt 2.8 bekannte) XML-Programmierschnittstelle DOM angelehnt sind, erlauben es dem Anwendungsprogramm, durch die Graphstruktur eines lokal vorliegenden Sequenzausschnitts zu navigieren. Bei dieser Navigation können die in den einzelnen Knoten enthaltenen Informationen ausgelesen und bei Bedarf verändert werden. Darüber hinaus besteht die Möglichkeit, Änderungen an der Graphstruktur des Sequenzausschnitts vorzunehmen — so lassen sich z. B. Knoten einfügen oder löschen.

Zur Illustration der Funktionsweise „DOM-artiger“ Sequenzausschnittsmethoden sind im Folgenden exemplarisch einige Methodenaufrufe aufgeführt.

- *sps(2)* — Zugriff auf den zweiten Sequenzausschnittsteil
- *tree(8)* — Navigation zum Wurzelknoten des achten Baums
- *v_child(3)* — Navigation zum drittlinkesten V-Kindknoten
- *e_child(“Artikel“)* — Navigation zum (linkesten) E-Kindknoten namens *Artikel*
- *getNodeName()* — Auslesen des Knotennamens
- *setVal(0.77)* — Setzen des Werts (eines V-Knotens) auf *0.77*
- *deleteNode()* — Löschen des Knotens

Die Nutzung entsprechender Sequenzausschnittsmethoden wurde im Abschnitt 4.2.7 bereits anhand eines konkreten Beispiels demonstriert. Für weitere Details verweisen wir auf [Ott07].

6.5.1.2 XQuery-Update-Facility-artige lokale Verarbeitung

Bei der Nutzung von „XQuery-Update-Facility-artigen“ Sequenzausschnittsmethoden werden die lokal vorzunehmenden Änderungen mit Hilfe von Ausdrücken spezifiziert, die in ihrer Syntax und Semantik an die XQuery-Update-Facility (Abschnitt 2.9) angelehnt[7] sind [Ott07]. Soll beispielsweise der im selben Geschäft einlösbare Gegenwert vom zweitlinkesten im Sequenzausschnitt enthaltenen Bonusangebot aufgewertet werden, so ist dies mit Hilfe des in *Abbildung 6.19* gezeigten Ausdrucks möglich, welcher der Sequenzausschnittsmethode *updateSP()*[8] als Parameter zu übergeben ist. Wie im Abschnitt 4.2.7 setzen wir hier einen zusammenhängenden (also aus einem einzigen Sequenzausschnittsteil bestehenden) Sequenzausschnitt voraus, bei dessen zweitlinkesten Baum es sich um das in Abbildung 3.30 dargestellte Bonusangebot handelt.

```
replace value of node sps(1)[2]/element(GegenWert)/atomVal()[1]
                with sps(1)[2]/element(GegenWert)/atomVal()[1] + 0.50
```

Abbildung 6.19: Ausdruck, angelehnt an die XQuery-Update-Facility

Innerhalb des in Abbildung 6.19 gezeigten Ausdrucks wird ein Pfadausdruck (Abschnitt 2.9) genutzt, um den zu ändernden V-Knoten zu identifizieren: Mittels *sps(1)[2]* wird zunächst der Wurzelknoten vom zweitlinkesten Baum des ersten (und gleichzeitig einzigen) Sequenzausschnittsteils ausgewählt. Ausgehend von diesem Wurzelknoten gelangt man durch den Lokalisierungsschritt *element(GegenWert)* dann zum (einzigen) E-Kindknoten namens *GegenWert*. Mit Hilfe des Lokalisierungsschritts *atomVal()[1]* wird anschließend der erste (also linkeste) V-Kindknoten dieses E-Knotens ermittelt.

6.5.2 Validitätsüberwachung

Eine XQuery-Sequenz kann bezüglich eines XML-Schemas *gültig* sein (Abschnitt 2.11.3). Beim (im Rahmen der Sequenzcursor-basierten Verarbeitung stattfindenden) Einbringen von lokalen Änderungen besteht grundsätzlich allerdings die Gefahr, dass eine solche Gültigkeit zerstört wird.

Durch eine lokale Validitätsüberwachung kann erreicht werden, dass (drohende) Gültigkeitsverletzungen z. T. bereits während der lokalen Verarbeitung des Sequenzausschnitts — also nicht erst beim Einbringen der lokalen Änderungen — erkannt werden. Zu den prinzipiell lokal erkennbaren Validitätsverletzungen gehören übrigens auch Verstöße gegen (möglicherweise) vom XML-Schema vorgegebene Schlüssel- bzw. Fremdschlüsselbedingungen (Abschnitt 2.7.3).

Änderungen an einem Knoten können zu einer Typverletzung des Vaterknotens und somit zu einer Validitätsverletzung führen. Dies gilt auch dann, wenn der Vaterknoten gar kein Bestandteil des lokal verarbeiteten Sequenzausschnitts ist. Die Validitätsverletzung ist in diesem Fall jedoch nicht lokal feststellbar — der betroffene Vaterknoten gehört ja nicht zum

[7] Da die lokale Verarbeitung der Sequenzausschnitte auf der typed-value-orientierten Repräsentation basiert, muss eine Anpassung/Erweiterung der „herkömmlichen“ Ausdrucksmöglichkeiten von XQuery-Update-Facility vorgenommen werden.

[8] Der Methodenname *updateSP* steht für *update sequence part*. Der die Änderung spezifizierende Ausdruck wird dieser Methode in Form einer Zeichenkette übergeben.

lokal vorliegenden Sequenzausschnitt. Dies zeigt, dass sich gewisse Validitätsverletzungen *nicht* lokal erkennen lassen. Wenn die Gültigkeit einer Sequenz garantiert werden soll, darf beim Einbringen der lokalen Änderungen trotz einer etwaigen lokalen Validitätsüberwachung also nicht auf eine (die gesamte Sequenz berücksichtigende) Validitätsprüfung verzichtet werden. Es ist möglich, eine solche Validitätsprüfung beim Einbringen der lokalen Änderungen zu veranlassen (Abschnitt 6.6.3).

Bezüglich einer lokalen Validitätsüberwachung kann zwischen den folgenden beiden Varianten unterschieden werden:

- *permanente lokale Validitätsüberwachung*

  Es wird verhindert, dass am Sequenzausschnitt Änderungen vorgenommen werden, die zu einer (lokal erkennbaren) Validitätsverletzung führen. Hierzu wird bei jedem ändernden Zugriff geprüft, ob dieser eine Verletzung der Validität hervorrufen würde. Wird dabei eine Validitätsverletzung erkannt, scheitert der entprechende Zugriffsversuch.

  Der Vorteil dieser Form der Validitätsüberwachung besteht darin, dass die Gültigkeit (abgesehen von lokal nicht feststellbaren Verstößen) zu jeder Zeit gewährleistet ist — lokal erkennbare Validitätsverletzungen werden gar nicht erst zugelassen. Diesem Vorteil steht jedoch der Nachteil gegenüber, dass die Validitätsüberwachung sehr aufwendig ist: Bei jeder einzelnen lokalen Änderung muss getestet werden, ob es zu einer Verletzung der Validität kommen würde.

- *lokale Validitätsüberwachung bei Bedarf*

  Am Sequenzausschnitt können (zwischenzeitlich) Änderungen vorgenommen werden, die eine Validitätsverletzung nach sich ziehen. Bei Bedarf (beispielsweise nachdem mehrere Änderungen durchgeführt wurden) kann das Anwendungsprogramm dann explizit eine lokale Prüfung der Validität veranlassen.

  Für diese Variante spricht vor allem, dass sich im Vergleich zu einer permanenten lokalen Validitätsüberwachung eine bessere Performance erzielen lässt, da bei ändernden Zugriffen nicht jedes Mal eine Validitätsprüfung erforderlich ist. Ein weiterer Vorteil besteht darin, dass das Anwendungsprogramm bei der lokalen Verarbeitung des Sequenzausschnitts die Flexibilität besitzt, Zwischenschritte vorzunehmen, die zu einer vorübergehenden Verletzung der Validität führen. Zu den Nachteilen dieser Form der lokalen Validitätsüberwachung zählt, dass (unbeabsichtigte) Validitätsverletzungen nicht sofort erkannt werden. Damit ist es für das Anwendungsprogramm i. d. R. schwieriger, geeignet auf derartige Validitätsverstöße zu reagieren.

Der Anwendungsprogrammierer sollte (in Abhängigkeit von den konkreten Erfordernissen der zu erstellenden Anwendung) flexibel entscheiden können, ob eine permanente lokale Validitätsüberwachung durchgeführt werden soll oder nicht. Dabei kann es auch sinnvoll sein, eine solche permanente Überwachung nur *zeitweise* vorzunehmen und die Validität ansonsten „bei Bedarf“ zu prüfen.

Zum Ein- und Ausschalten einer permanenten lokalen Validitätsüberwachung stehen die Sequenzausschnittsmethoden *startValidityCheck()* bzw. *stopValidityCheck()* zur Verfügung. Eine Validitätsprüfung „bei Bedarf“ ist mit Hilfe der Sequenzausschnittsmethode *checkValidity()* möglich, welche als Resultat einen Wahrheitswert zurückliefert.

Die Prüfung der Gültigkeit erfolgt normalerweise bezüglich des XML-Schemas, das beim Definieren des Sequenzausschnitts spezifiziert wurde (Abschnitt 6.3). Falls erforderlich, (z. B. wenn bei der Sequenzausschnittsdefinition kein XML-Schema festgelegt wurde) kann beim Aufruf der Methoden *startValidityCheck()* bzw. *checkValidity()* ein XML-Schema angegeben werden. Dieses XML-Schema wird dann für die lokale Gültigkeitsprüfung genutzt.

6.5.3 Änderungsprotokollierung

Änderungen, die (mit Hilfe von Sequenzausschnittsmethoden) am Sequenzausschnitt vorgenommen werden, werden lokal automatisch protokolliert. Die Art und Weise dieser Änderungsprotokollierung ist für den Anwendungsprogrammierer transparent und wird nicht vom Sequenzcursor-basierten Verarbeitungsmodell vorgegeben.

Die konkrete Durchführung der lokalen Änderungsprotokollierung kann (und sollte) auf die verwendete Programmiersprachenrepräsentation der Sequenzausschnitte abgestimmt sein. Wird z. B. jeder Knoten als separates Objekt repräsentiert (Abschnitt 6.4.1), so bietet es sich an, die Änderungsprotokollierung durch ein Markieren der Objekte zu realisieren: Beim Einfügen, Ändern oder Löschen eines Knotens wird das korrespondierende Objekt geeignet markiert. Wird beispielsweise ein Knoten gelöscht, so erfolgt kein Löschen des ihn repräsentierenden Objekts — das entsprechende Objekt wird lediglich als „gelöscht“ gekennzeichnet. Für weitere Details verweisen wir auf [Ott07].

Beim Einbringen der lokalen Änderungen werden (auf Anforderung) sämtliche den Sequenzausschnitt betreffende Änderungsinformationen in einem zusammenhängenden lokalen Speicherbereich bereitgestellt, aus dem sie dann vom SQL-Laufzeitsystem ausgelesen werden (Abschnitt 4.2.8.1). Diese Bereitstellung von Änderungsinformationen erfolgt, indem die während der Änderungsprotokollierung gesammelten Informationen ausgewertet und in eine (vom SQL-Laufzeitsystem verarbeitbare) Repräsentation überführt werden. Die Details dieser Repräsentation sind für den Anwendungsprogrammierer transparent und werden vom Sequenzcursor-basierten Verarbeitungsmodell nicht näher spezifiziert.

Eine Möglichkeit für eine geeignete Repräsentation der (im lokalen Speicherbereich bereitzustellenden) Änderungsinformationen besteht darin, eine Folge so genannter *Änderungseinträge* zu nutzen. Jeder dieser Änderungseinträge korrespondiert dabei jeweils mit einem geänderten, gelöschten oder neu eingefügten Knoten und enthält alle für das Ändern, Löschen bzw. Einfügen dieses Knotens erforderlichen Informationen. Nähere Einzelheiten hierzu finden sich in [Jat07].

Um das (eigentliche) Einbringen der lokalen Änderungen effizient durchführen zu können und den mit der Übertragung der Änderungsinformationen verbundenen Aufwand zu begrenzen, sollten die dem SQL-Laufzeitsystem lokal bereitgestellten Änderungsinformationen möglichst *kompakt* sein. Dies bedeutet, dass u. a. die folgenden „Kompaktheits-Forderungen“ erfüllt werden sollten:

- Mehrere Änderungen, die ein und denselben Knoten betreffen, werden zu *einer* „resultierenden“ („akkumulierten“) Änderung zusammengefasst.
- Änderungen an einem Knoten, der im weiteren Verlauf der lokalen Verarbeitung gelöscht wird, sind zu ignorieren.

- Wird während der lokalen Verarbeitung ein Knoten neu eingefügt und anschließend wieder gelöscht, so darf sich weder das Einfügen noch das Löschen in den Änderungsinformationen widerspiegeln.

In [Jat07] und [Ott07] konnte gezeigt werden, dass eine Erfüllung der zuvor erwähnten „Kompaktheits-Forderungen“ möglich ist, wenn eine auf Markierungen basierende Änderungsprotokollierung durchgeführt wird und die (dem SQL-Laufzeitsystem lokal bereitzustellenden) Änderungsinformationen als Folge von Änderungseinträgen repräsentiert werden. Für Details diesbezüglich verweisen wir auf die beiden genannten Literaturquellen.

6.6 Einbringen von lokalen Änderungen

Im Abschnitt 4.2.8.1 haben wir geschildert, welche Schritte durchgeführt werden müssen, um das Einbringen der lokalen Änderungen vorzubereiten. Das eigentliche Einbringen der lokalen Änderungen wurde dann mit Hilfe der Sequenzcursoranweisung `BRING IN LOCAL CHANGES` veranlasst.

Im Abschnitt 6.6.1 werden wir darauf eingehen, dass das (eigentliche) Einbringen der lokalen Änderungen aus zwei Phasen besteht. Abschnitt 6.6.2 widmet sich anschließend den Voraussetzungen, die für ein erfolgreiches Änderungseinbringen vorliegen müssen. Die beim Einbringen der lokalen Änderungen spezifizierbaren Validitätsprüfungsmodi sind Gegenstand von Abschnitt 6.6.3.

6.6.1 2-Phasigkeit des Einbringens

Beim Einbringen der lokalen Änderungen lassen sich die folgenden beiden Phasen unterscheiden.

- *1. Phase: Ermittlung der am Anfrageergebnis vorzunehmenden Änderungen*

  Die Änderungsinformationen, die vom SQL-Laufzeitsystem aus dem lokalen Speicherbereich ausgelesen werden (vgl. Abschnitt 4.2.8.1), sind auf den lokal verarbeiteten Sequenzausschnitt bezogen. Das SQL-Laufzeitsystem nutzt diese Änderungsinformationen, um zu ermitteln, welche Änderungen an der dem Sequenzausschnitt zugrunde liegenden (im Anfrageergebnis enthaltenen) Sequenz vorzunehmen sind.

  Die Überführung der Sequenzausschnitts-bezogenen Änderungsinformationen in Sequenz-bezogene Änderungsinformationen ist in den meisten Fällen trivial: Wurde lokal beispielsweise der Inhalt eines C-Knotens geändert, so ist diese Änderung einfach auf den entsprechenden C-Knoten der im Anfrageergebnis enthaltenen Sequenz zu übertragen. Es existieren jedoch auch Situationen, in denen die Überführung der Änderungsinformationen komplizierter ist. So kann z. B. das lokale Löschen *eines* Knotens das Löschen *mehrerer* Knoten der Sequenz nach sich ziehen. Dies ist dann der Fall, wenn das Löschen des entsprechenden Knotens zum automatischen Löschen eines Teilbaums führt, der bei der Definition des Sequenzausschnitts unter Nutzung der Option `ON DELETE FORCE` ausgeschlossen wurde (vgl. Abschnitt 6.3.1).

- *2. Phase: Rückabbildung der Änderungen auf die Basistabelle*

  Die (in der 1. Phase ermittelten) am Anfrageergebnis vorzunehmenden Änderungen werden vom SQL-Laufzeitsystem auf die dem Anfrageergebnis zugrunde liegende Basistabelle rückabgebildet. Dies bedeutet, dass die an der entsprechenden Sequenz des Anfrageergebnisses vorzunehmenden Änderungen „1-zu-1“ auf die mit dieser Sequenz korrespondierende (in der Basistabelle enthaltene) Ursprungssequenz angewendet werden. Eine solche „1-zu-1-Anwendung“ der Änderungen ist möglich, da sich — ein *änderndes* lokales Arbeiten vorausgesetzt — eine im Anfrageergebnis enthaltene Sequenz nicht von der ihr zugrunde liegenden Ursprungssequenz unterscheidet (vgl. Abschnitt 4.4).

6.6.2 Voraussetzungen für Einbringbarkeit

Ein erfolgreiches Einbringen der lokalen Änderungen ist genau dann möglich, wenn sich beide Phasen des Einbringens mit Erfolg durchführen lassen. Wir werden im Folgenden darauf eingehen, welche Voraussetzungen für die Durchführbarkeit der beiden Phasen erfüllt sein müssen.

- *Voraussetzungen für Durchführbarkeit der 1. Phase*

  Die lokal am Sequenzausschnitt vorgenommenen Änderungen müssen sich erfolgreich auf die dem Sequenzausschnitt zugrunde liegende Sequenz übertragen lassen. Dies bedeutet insbesondere, dass es nicht notwendig werden darf, einen Knoten zu löschen, der einen Kindknoten besitzt, bei dem es sich um den Wurzelknoten eines Teilbaums handelt, welcher beim Definieren des Sequenzausschnitts unter Nutzung der Option `ON DELETE RESTRICT` ausgeschlossen wurde (vgl. Abschnitt 6.3.1).

- *Voraussetzungen für Durchführbarkeit der 2. Phase*

  In der 2. Phase des Einbringens der lokalen Änderungen werden die an der entsprechenden (im aktuellen Ergebnistupel enthaltenen) Sequenz vorzunehmenden Änderungen „1-zu-1“ auf die mit dieser Sequenz korrespondierende Ursprungssequenz angewendet. Da eine solche „1-zu-1-Anwendung“ der Änderungen (im Wesentlichen) der Durchführung eines positionierten `UPDATE`s (Abschnitt 2.4.3) entspricht, müssen dabei dieselben Voraussetzungen erfüllt sein, als würde ein positioniertes `UPDATE` vorgenommen. So darf beispielsweise die zur Erzeugung des Anfrageergebnisses verwendete SQL-`SELECT`-Anfrage keine `DISTINCT`-Klausel enthalten (vgl. Abschnitt 2.4.3).

6.6.3 Validitätsprüfungsmodi

Da sich gewisse Validitätsverletzungen nicht lokal erkennen lassen, ist eine lokale Validitätsüberwachung kein Ersatz für eine Validitätsprüfung, die beim Einbringen der lokalen Änderungen durchgeführt wird und die gesamte Sequenz berücksichtigt (Abschnitt 6.5.2). Ob eine derartige, umfassende Validitätsprüfung allerdings überhaupt erwünscht bzw. sinnvoll ist, hängt von den konkreten Erfordernissen der betreffenden Anwendung ab.

Gemäß des Sequenzcursor-basierten Verarbeitungsmodells kann frei (und flexibel) entschieden werden, ob beim Einbringen der lokalen Änderungen eine Gültigkeitsprüfung vorgenommen werden soll oder nicht: Beim Veranlassen des Änderungseinbringens ist es

möglich, einen der beiden im Folgenden aufgeführten Validitätsprüfungsmodi zu spezifizieren.[9] Einzelheiten zur dabei zu verwendenden Syntax finden sich im Abschnitt 7.3.5.

- `WITH VALIDITY CHECK`

  Das Einbringen der lokalen Änderungen scheitert, falls es zu einer Verletzung der Validität führen würde. Zu den Gültigkeitsverstößen, die ein Scheitern des Änderungseinbringens zur Folge haben, zählen u. a. übrigens auch Verletzungen etwaiger vom XML-Schema vorgegebener Schlüssel- bzw. Fremdschlüsselbedingungen (Abschnitt 2.7.3).

  Dieser Validitätsprüfungsmodus besitzt den Vorteil, dass die Gültigkeit einer XQuery-Sequenz nicht durch das Einbringen von lokalen Änderungen zerstört werden kann — die Gültigkeit ist somit „garantiert“ auch noch im Anschluss an ein Änderungseinbringen gewährleistet.

  Die Validitätsprüfung erfolgt i. d. R. bezüglich des XML-Schemas, das bei der Sequenzausschnittsdefinition festgelegt wurde (Abschnitt 6.3). Bei Bedarf (z. B. wenn beim Definieren des Sequenzausschnitts kein XML-Schema spezifiziert wurde) kann beim Einbringen der lokalen Änderungen ein XML-Schema angegeben werden. Die Prüfung der Validität wird dann basierend auf diesem XML-Schema vorgenommen.

- `WITHOUT VALIDITY CHECK`

  Beim Einbringen der lokalen Änderungen wird keine Gültigkeitsprüfung durchgeführt. Die Änderungen werden also auch dann eingebracht, wenn dadurch Validitätsverletzungen entstehen.

  Dieser Modus ist beispielsweise zu verwenden, falls gar kein XML-Schema existiert, bezüglich dessen die (dem Sequenzausschnitt zugrunde liegende) XQuery-Sequenz gültig ist bzw. gültig sein soll — XQuery-Sequenzen müssen ja nicht zwangsläufig ein zugeordnetes XML-Schema besitzen.

6.7 Berücksichtigung des Konzepts der Knotenidentität

Wie im Abschnitt 2.11.3 geschildert, berücksichtigt die SQL-Norm das Konzept der Knotenidentität, indem sie (für die Übergabe von XQuery-Sequenzen) die beiden Übergabemechanismen *BY REF* und *BY VALUE* anbietet. Wir werden im Folgenden anhand eines kleinen Beispiels zeigen, dass bei der im Rahmen des Sequenzcursor-basierten Verarbeitungsablaufs stattfindenden „Übergabe“ von Sequenzausschnitten ans Anwendungsprogramm — also der Übertragung der Sequenzausschnitte ins Anwendungsprogramm — der Übergabemechanismus *BY VALUE* (und *nicht* die Variante *BY REF*) genutzt wird.

Für unsere Betrachtungen setzen wir den im rechten Teil der *Abbildung 6.20* dargestellten (aus drei Knoten bestehenden) Sequenzausschnitt voraus, der bereits ins Anwendungsprogramm übertragen worden sei. Die diesem Sequenzausschnitt zugrunde liegende (im Anfrageergebnis enthaltene) Sequenz wird im linken Teil der Abbildung veranschaulicht.

Während der in der Sequenz enthaltene Knoten *16* einen Vaterknoten besitzt, ist der Knoten *16* des (lokal vorliegenden) Sequenzausschnitts vaterlos. Die beiden Knoten sind somit

[9] Wird kein Validitätsprüfungsmodus spezifiziert, wird der Modus `WITHOUT VALIDITY CHECK` genutzt.

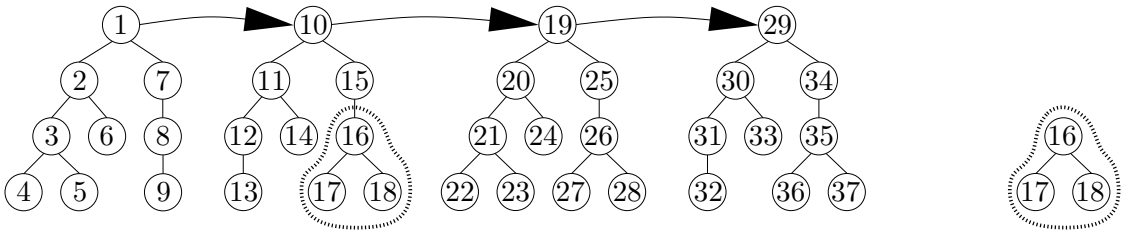


Abbildung 6.20: Sequenz und lokal vorliegender Sequenzausschnitt

nicht identisch, sondern besitzen verschiedene Knotenidentitäten.[10] Damit hat der Knoten *17* der Sequenz einen anderen Vater als der Knoten *17* des Sequenzausschnitts, sodass auch diese beiden Knoten nicht identisch sind. Gleiches gilt für die beiden Knoten *18*.

Bei den im lokal vorliegenden Sequenzausschnitt enthaltenen Knoten handelt es sich also um *Kopien* der Knoten der Sequenz (und nicht um die Knoten der Sequenz selbst). Dies zeigt, dass bei der Übertragung des Sequenzausschnitts der Übergabemechanismus *BY VALUE* verwendet wurde.

[10] Ein und derselbe Knoten kann nicht gleichzeitig einen Vaterknoten besitzen und vaterlos sein.

Kapitel 7

Vorschlag zur Erweiterung von SQL

Die zur Unterstützung der Sequenzcursor-basierten Verarbeitung eingeführten Sequenzcursoranweisungen wurden von uns als (mögliche) Erweiterung der Datenbanksprache SQL konzipiert. Gleiches gilt für die zur Kommunikation zwischen dem Anwendungsprogramm und dem SQL-Laufzeitsystem dienenden Datenstrukturen SQLSCFA und SQLSPCA.

Wir werden im aktuellen Kapitel näher auf die Syntax der Sequenzcursoranweisungen und den Aufbau der beiden Datenstrukturen eingehen. Das vorliegende Kapitel beschreibt also die Sprachkonstrukte, um die SQL (unserer Ansicht nach) erweitert werden sollte. Der Vorteil einer solchen SQL-Erweiterung bestände darin, dass SQL damit um Funktionalität ergänzt würde, die es ermöglicht, in Anfrageergebnissen enthaltene XML-Werte adäquat und komfortabel zu verarbeiten.

Sämtliche Sequenzcursoranweisungen sind für eine Nutzung in dynamischem Embedded SQL geeignet [Böh07]. Dies bedeutet, dass alle Sequenzcursoranweisungen flexibel und dynamisch zur Laufzeit zusammengebaut (und ausgeführt) werden können. Alternativ ist es natürlich auch möglich, die Sequenzcursoranweisungen (gemäß der Vorgehensweise bei statischem Embedded SQL) fest im Quellcode zu „verdrahten“. Für eine Gegenüberstellung beider Einbettungsvarianten verweisen wir auf Abschnitt 2.3.1.

Da sich Syntaxdiagramme gegenüber anderen Darstellungsformen durch eine einfache Lesbarkeit und Verständlichkeit auszeichnen, werden wir die Syntax der Sequenzcursoranweisungen mit Hilfe von Syntaxdiagrammen spezifizieren. Eine Erklärung der dabei verwendeten Notation[1] erfolgt im Abschnitt 7.1. Im Anschluss stellt Abschnitt 7.2 diejenigen Sequenzcursoranweisungen vor, die sich unmittelbar auf Sequenzcursor beziehen. Abschnitt 7.3 widmet sich dann den Sequenzausschnitts-bezogenen Sequenzcursoranweisungen. Die zur Kommunikation zwischen Anwendungsprogramm und SQL-Laufzeitsystem dienenden Datenstrukturen sind abschließend Gegenstand von Abschnitt 7.4.

[1] Zur Darstellung der Syntaxdiagramme nutzen wir dieselbe Notation wie bereits in [Mül03]. Diese Darstellungsform wird beispielsweise auch in [Gol05] verwendet.

7.1 Notation zur Darstellung von Syntaxdiagrammen

Die Syntaxdiagramme sind (wie üblich) der Linie folgend von oben nach unten und auf einer Ebene jeweils von links nach rechts zu lesen *(Abbildung 7.1)*. Der Anfang einer Anweisung wird durch `>>` und das Ende einer Anweisung durch `><` markiert. Das Symbol `->` drückt aus, dass die Anweisung auf der nächsten Zeile fortgesetzt wird. Entsprechend kennzeichnet `>-` eine Fortsetzung der Anweisung von der vorangegangenen Zeile.

Schlüsselwörter (z. B. `CREATE`) werden durchgehend mit Großbuchstaben dargestellt. Für Variablen (z. B. `row-cursor-name` oder `sequence-part`) werden ausschließlich Kleinbuchstaben und das Symbol `-` (Bindestrich) verwendet. Das Symbol `-` (Bindestrich) tritt dabei nur *innerhalb* eines Variablennamens auf und auch dort nicht mehrfach direkt nacheinander, da ein mehrfaches Vorkommen dieses Symbols (z. B. `--`) als Trennung benutzt wird.

Wahlmöglichkeiten, Optionen und Wiederholungen werden (wie allgemein üblich) mit Hilfe von Verzweigungen dargestellt (Abbildung 7.1). Existiert bei mehreren optionalen Wahlmöglichkeiten eine Default-Möglichkeit, so wird diese im Syntaxdiagramm dadurch hervorgehoben, dass der entsprechende Pfad *oberhalb* der Hauptlinie verläuft.

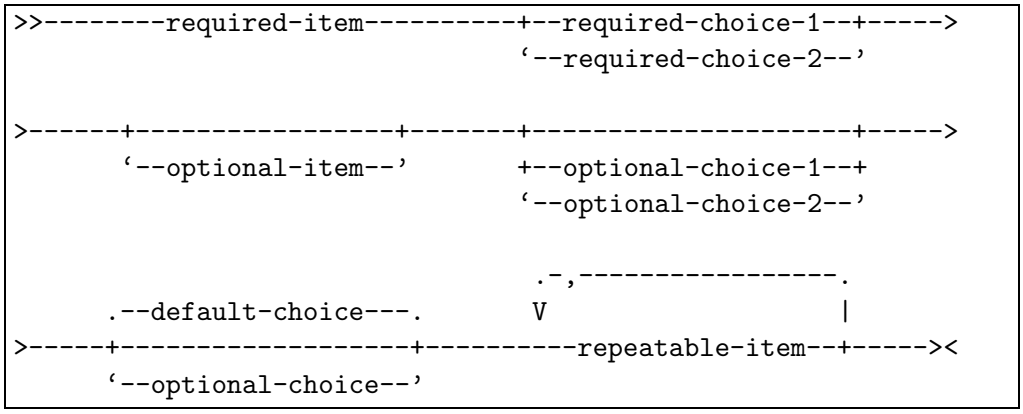


Abbildung 7.1: Verzweigungen im Syntaxdiagramm

Teile eines Syntaxdiagramms, die wiederum durch ein eigenes Syntaxdiagramm dargestellt werden, sind von den Symbolen `-|` und `|-` eingeschlossen. Wird beispielsweise durch ein separates Syntaxdiagramm definiert, wofür die Variable `frequently-used-item` steht *(Abbildung 7.2)*, so ist dies im übergeordneten Syntaxdiagramm *(Abbildung 7.3)* durch `-| frequently-used-item |-` dargestellt. Diese Vorgehensweise erlaubt es, Anweisungsteile außerhalb des globalen Syntaxdiagramms zu definieren, um so eine höhere Lesbarkeit und Übersichtlichkeit zu erreichen. Zudem eröffnet sich die Möglichkeit, häufig verwendete Anweisungsteile lediglich *einmal* zu definieren, um sie dann in verschiedenen Syntaxdiagrammen verwenden zu können.

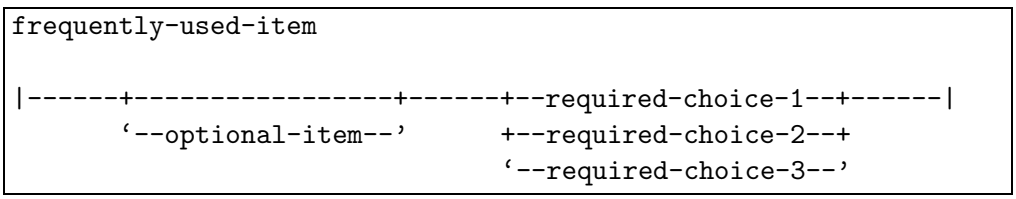


Abbildung 7.2: Syntaxdiagramm für `frequently-used-item`

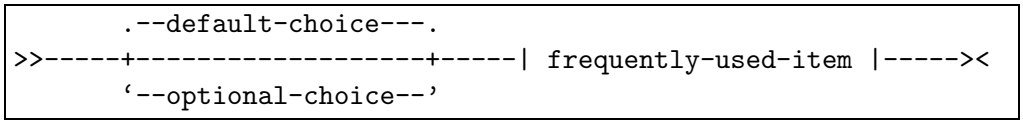


Abbildung 7.3: Syntaxdiagramm mit Verweis auf separates Syntaxdiagramm

7.2 Sequenzcursor-bezogene Anweisungen

In den sich hier anschließenden Abschnitten 7.2.1 bis 7.2.4 werden diejenigen Sequenzcursoranweisungen beschrieben, die unmittelbar auf Sequenzcursor (also nicht auf Sequenzausschnitte) bezogen sind. Detaillierte Informationen über Sequenzcursor bzw. über die Verwendung von Sequenzcursorn finden sich in den Kapiteln 4 und 5.

Jeder der Beschreibung einer Sequenzcursoranweisung dienende Abschnitt enthält außer einem (die Anweisungssyntax spezifizierenden) Syntaxdiagramm auch ein Beispiel für die Nutzung der entsprechenden Sequenzcursoranweisung sowie eine Erläuterung sämtlicher im Syntaxdiagramm vorkommenden Optionen und Parameter.

7.2.1 CREATE SEQUENCE CURSOR[S]

Die Anweisung `CREATE SEQUENCE CURSOR[S]` *(Abbildung 7.4)* dient zum Erzeugen von Sequenzcursorn.

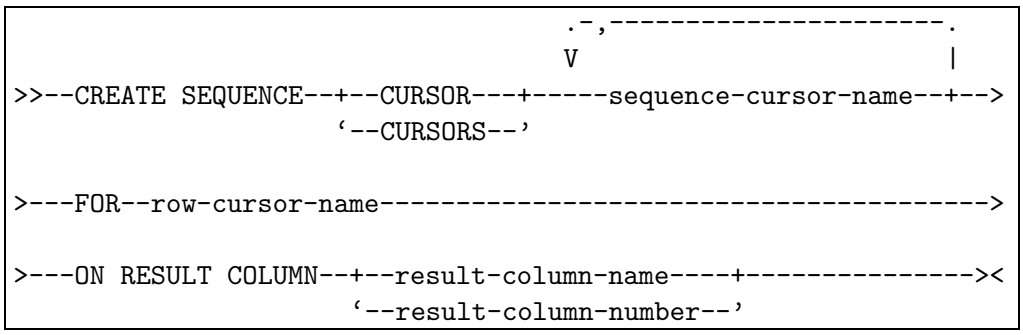


Abbildung 7.4: Syntaxdiagramm der Anweisung `CREATE SEQUENCE CURSOR[S]`

Beschreibung der Optionen und Parameter

`CURSOR`, `CURSORS`
: Es ist unerheblich, welches dieser beiden Schlüsselwörter genutzt wird.[2]

`sequence-cursor-name`
: Mit Hilfe dieses Parameters werden die (paarweise verschiedenen) Namen der zu erzeugenden Sequenzcursor übergeben. Es darf noch kein Sequenzcursor mit einem dieser Namen existieren.

[2] Durch die Wahlmöglichkeit zwischen `CURSOR` und `CURSORS` soll lediglich gewährleistet werden, dass die Anweisungen zum Erzeugen von Sequenzcursorn ohne Verstöße gegen die englische Grammatik formulierbar sind.

`row-cursor-name`
: Mittels dieses Parameters wird der Name des Tupelcursors übergeben, der den Sequenzcursorn zugeordnet werden soll. Durch die Angabe des Tupelcursornamens wird festgelegt, für welches Anfrageergebnis die Sequenzcursor zur Verfügung stehen sollen. Es muss einen geöffneten Tupelcursor mit dem entsprechenden Namen geben.

`result-column-name`
: Dieser Platzhalter steht für den Namen der Ergebnisspalte, an welche die Sequenzcursor gebunden werden sollen. Im (mit Hilfe des Tupelcursornamens identifizierten) Anfrageergebnis muss genau eine XML-Spalte mit dem entsprechenden Namen existieren.

`result-column-number`
: Dieser Parameter übergibt die (Spalten-)Nummer derjenigen Ergebnisspalte, an die die Sequenzcursor gebunden werden sollen. Das Anfrageergebnis muss eine Spalte mit der entsprechenden Nummer besitzen. Bei dieser Spalte muss es sich um eine XML-Spalte handeln.

Beispiel

```
CREATE SEQUENCE CURSORS meinSC1, meinSC2, meinSC3
FOR unserTC
ON RESULT COLUMN 21;
```

Abbildung 7.5: Beispiel für die Anweisung `CREATE SEQUENCE CURSOR[S]`

Mittels der Beispielanweisung *(Abbildung 7.5)* werden die Sequenzcursor *meinSC1*, *meinSC2* und *meinSC3* erzeugt. Diese drei Sequenzcursor werden dabei an die 21. Spalte des mit Hilfe des Tupelcursors *unserTC* erstellten Anfrageergebnisses gebunden.

7.2.2 DROP SEQUENCE CURSOR[S]

Die Anweisung `DROP SEQUENCE CURSOR[S]` *(Abbildung 7.6)* wird genutzt, um Sequenzcursor zu löschen.

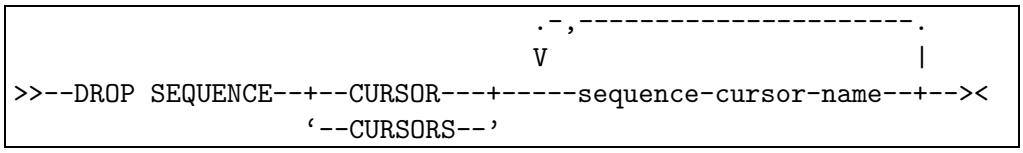


Abbildung 7.6: Syntaxdiagramm der Anweisung `DROP SEQUENCE CURSOR[S]`

Beschreibung der Optionen und Parameter

`CURSOR`, `CURSORS`
: Es ist irrelevant, welches dieser beiden Schlüsselwörter verwendet wird.

`sequence-cursor-name`
Dieser Parameter dient dazu, die (paarweise verschiedenen) Namen der zu löschenden Sequenzcursor zu übergeben. Es müssen Sequenzcursor mit den entsprechenden Namen vorhanden sein.

Beispiel

```
DROP SEQUENCE CURSORS meinSC1, meinSC2, meinSC3;
```

Abbildung 7.7: Beispiel für die Anweisung `DROP SEQUENCE CURSOR[S]`

Mit Hilfe der Beispielanweisung *(Abbildung 7.7)* werden die Sequenzcursor *meinSC1*, *meinSC2* und *meinSC3* gelöscht.

7.2.3 MOVE SEQUENCE CURSOR

Zum Positionieren eines Sequenzcursors steht die Anweisung `MOVE SEQUENCE CURSOR` *(Abbildung 7.8)* zur Verfügung. Nach jeder (erfolgreichen) Ausführung dieser Anweisung wird die SQLSCFA (Abschnitt 7.4.1) automatisch aktualisiert.

Beschreibung der Optionen und Parameter

`SEQUENCE CURSOR`
Die Angabe dieser beiden Schlüsselwörter ist optional und hat keinen Einfluss auf die Semantik.

`sequence-cursor-name`
Für diesen Platzhalter ist der Name des zu positionierenden Sequenzcursors einzusetzen. Es muss ein Sequenzcursor mit dem entsprechenden Namen existieren.

sonstige Optionen und Parameter
Die Bedeutung aller sonstigen im Syntaxdiagramm enthaltenen Optionen und Parameter ergibt sich aus den Erläuterungen des Abschnitts 5.2.

Wie in den Abschnitten 5.2.3 bis 5.2.5 erwähnt, kann nicht jede der im Abschnitt 5.2.2 vorgestellten (grundlegenden) Positionierungsmöglichkeiten um die Angabe einer Knotenart (z. B. `OF KIND TEXT`), eines Knotentyps (`OF TYPE node-type`) bzw. eines Knotennamens (`OF NAME node-name`[3]) erweitert werden. *Abbildung 7.9* fasst zusammen, bei welchen Positionierungsvarianten eine Festlegung von Knotenart, Knotentyp bzw. Knotenname möglich ist.

Wie im Abschnitt 5.2.6 geschildert, unterliegt die gleichzeitige Angabe von Knotenart und Knotentyp bzw. Knotenname gewissen Beschränkungen. *Abbildung 7.10* kann

[3] Die Parameter `node-type` und `node-name` dienen als Platzhalter für den entsprechenden Knotentyp bzw. Knotennamen.

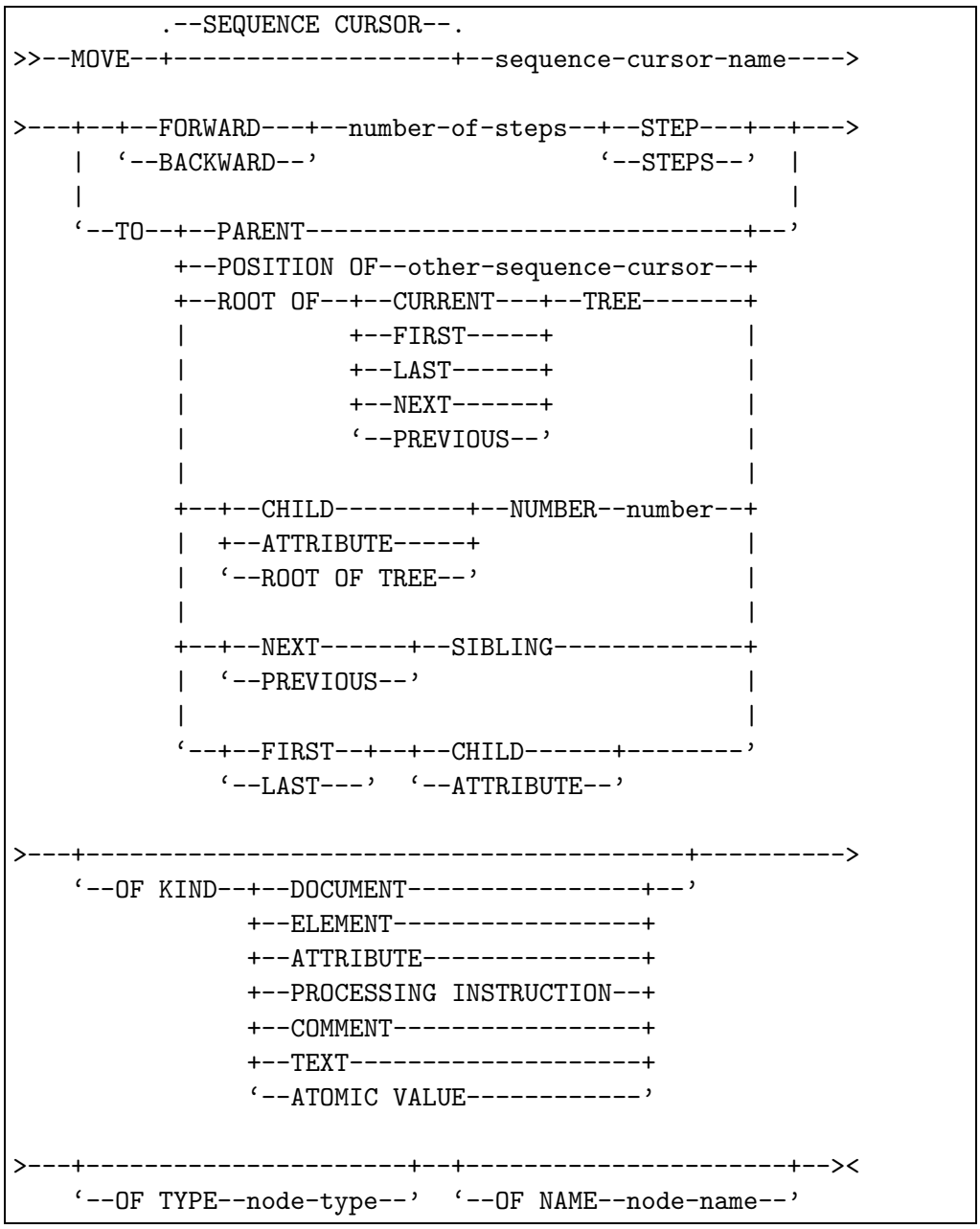


Abbildung 7.8: Syntaxdiagramm der Anweisung `MOVE SEQUENCE CURSOR`

entnommen werden, welche Möglichkeiten es gibt, die Angabe einer bestimmten Knotenart mit der Festlegung eines Knotentyps bzw. Knotennamens zu kombinieren.

Beispiel

Die Ausführung der in *Abbildung 7.11* gezeigten Beispielanweisung bewirkt, dass der Sequenzcursor *meinSC1* direkt zum ersten sich auf Wurzelebene befindenden E-Knoten *BonusAngebot* vom Typ *BATyp* springt.

Wir werden die Semantik der Festlegung einer Knotenart, eines Knotentyps bzw. eines

genutzte Positionierungsvariante	Knotenart spezifizierbar?	Knotentyp spezifizierbar?	Knotenname spezifizierbar?
`TO POSITION OF ...`	–	–	–
`TO PARENT`			
`TO ROOT OF CURRENT TREE`			
`TO FIRST/LAST ATTRIBUTE`	–	✓	✓
`TO ATTRIBUTE NUMBER ...`			
`TO NEXT/PREVIOUS SIBLING`	✓[1]	✓	✓
`TO FIRST/LAST CHILD`			
`TO CHILD NUMBER ...`			
sonstige	✓	✓	✓

✓ = ja, – = nein

[1]Die Knotenarten `DOCUMENT` und `ATTRIBUTE` sind *nicht* spezifizierbar.

Abbildung 7.9: Spezifizierbarkeit von Knotenart, Knotentyp und Knotenname

spezifizierte Knotenart	Knotentyp spezifizierbar?	Knotenname spezifizierbar?
`ELEMENT`	✓	✓
`ATTRIBUTE`		
`ATOMIC VALUE`	✓	–
`PROCESSING INSTRUCTION`	–	✓[1]
sonstige	–	–

[1]Das Verarbeitungsanweisungsziel wird als Knotenname aufgefasst (Abschnitt 2.10.2.3).

Abbildung 7.10: Spezifizierbarkeit von Knotentyp und Knotenname

```
MOVE SEQUENCE CURSOR meinSC1 TO ROOT OF FIRST TREE
OF KIND ELEMENT OF TYPE BATyp OF NAME BonusAngebot;
```

Abbildung 7.11: Beispiel für die Anweisung `MOVE SEQUENCE CURSOR`

Knotennamens im Folgenden allgemeingültig (also ohne Beschränkung auf konkrete Beispiele) beschreiben.

Semantik der Angabe von Knotenart, Knotentyp bzw. Knotenname

Enthält eine Positionierungsanweisung die Angabe einer Knotenart, eines Knotentyps bzw. eines Knotennamens, so erfolgt ihre Ausführung (zumindest aus logischer Sicht) in zwei Schritten:

- *1. Schritt:*

  Sämtliche Knoten, die von der Vorgabe der Knotenart, des Knotentyps bzw. des Knotennamens abweichen, werden ausgeblendet. Liegt allerdings eine *relative* Positionierung vor und befindet sich der betreffende Sequenzcursor zudem im Status *„positioniert“*, so werden der Knoten, auf dem der Sequenzcursor aktuell steht, sowie dessen (möglicherweise existierender) Vaterknoten auch dann *nicht* ausgeblendet,

wenn sie die Vorgaben bezüglich Knotenart, Knotentyp bzw. Knotenname verletzen sollten.[4]

Setzen wir die in *Abbildung 7.12* veranschaulichte Sequenz als Ausgangsbasis für die in Abbildung 7.11 dargestellte Beispielanweisung voraus, so ergibt sich das in *Abbildung 7.13* gezeigte Zwischenergebnis.

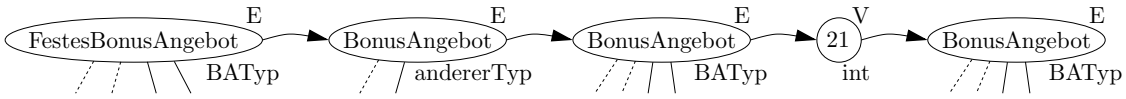


Abbildung 7.12: Beispielsequenz (vereinfacht)

Abbildung 7.13: Zwischenergebnis nach Schritt 1

- *2. Schritt:*

  Die Positionierung des Sequenzcursors erfolgt auf dem im 1. Schritt ermittelten Zwischenergebnis, als wäre *keine* Vorgabe bezüglich Knotenart, Knotentyp bzw. Knotenname vorhanden. Anschließend werden alle ausgeblendeten Knoten wieder eingeblendet.

  In unserem Beispiel wird also die Positionierungsanweisung `MOVE SEQUENCE CURSOR meinSC1 TO ROOT OF FIRST TREE` basierend auf dem in Abbildung 7.13 dargestellten Zwischenergebnis ausgeführt. Bezogen auf die ursprüngliche Sequenz wird der Sequenzcursor *meinSC1* damit auf den Wurzelknoten des drittlinkesten Baums positioniert. Bei diesem Knoten handelt es sich um den linkesten in der Wurzelebene enthaltenen E-Knoten, der den Namen *BonusAngebot* besitzt und vom Typ *BATyp* ist.

7.2.4 `GET INFORMATION ABOUT SEQUENCE CURSOR`

Die Anweisung `GET INFORMATION ABOUT SEQUENCE CURSOR` *(Abbildung 7.14)* dient dazu, Informationen über einen bestimmten Sequenzcursor abzurufen. Die Ausführung dieser Anweisung bewirkt eine Aktualisierung der SQLSCFA.

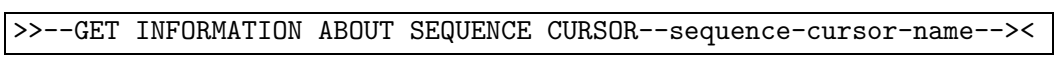


Abbildung 7.14: Syntaxdiagramm für `GET INFORMATION ABOUT SEQUENCE CURSOR`

[4] Hierdurch wird bei *relativen* Positionierungen sichergestellt, dass die „Startposition“ des Sequenzcursors erhalten bleibt. Außerdem wird verhindert, dass Geschwisterbeziehungen zerstört werden, die (gegebenenfalls) für die Positionierung wesentlich sind.

Beschreibung der Optionen und Parameter

`sequence-cursor-name`
: Dieser Platzhalter steht für den Namen desjenigen Sequenzcursors, über den die Informationen abgerufen werden sollen. Es muss ein Sequenzcursor mit dem entsprechenden Namen vorhanden sein.

Beispiel

```
GET INFORMATION ABOUT SEQUENCE CURSOR meinSC1;
```

Abbildung 7.15: Beispiel für `GET INFORMATION ABOUT SEQUENCE CURSOR`

Die in *Abbildung 7.15* dargestellte Beispielanweisung wird genutzt, um Informationen über den Sequenzcursor *meinSC1* abzufragen.

7.3 Sequenzausschnitts-bezogene Anweisungen

Nachdem im Abschnitt 7.2 die Sequenzcursor-bezogenen Sequenzcursoranweisungen vorgestellt wurden, widmen sich die nun folgenden Abschnitte 7.3.1 bis 7.3.6 den verbleibenden (d. h. Sequenzausschnitts-bezogenen) Sequenzcursoranweisungen. Für nähere Einzelheiten über Sequenzausschnitte bzw. die Nutzung von Sequenzausschnitten verweisen wir auf die Kapitel 4 und 6.

7.3.1 `DEFINE SEQUENCE PART`

Die Anweisung `DEFINE SEQUENCE PART` *(Abbildung 7.16)* wird verwendet, um einen Sequenzausschnitt zu definieren.

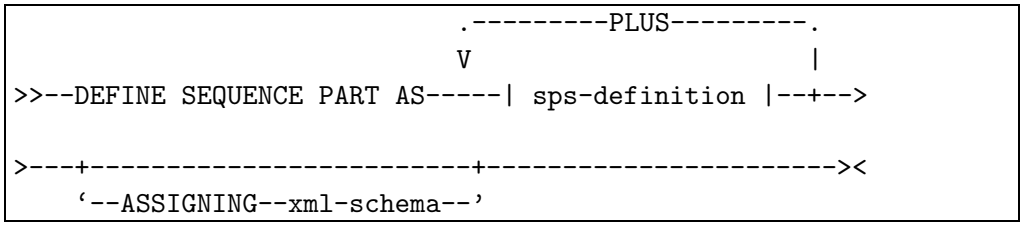


Abbildung 7.16: Syntaxdiagramm der Anweisung `DEFINE SEQUENCE PART`

Beschreibung der Optionen und Parameter

`PLUS`
: Besteht der zu definierende Sequenzausschnitt aus mehreren Sequenzausschnittsteilen, so werden die Definitionen der einzelnen Sequenzausschnittsteile durch das Schlüsselwort `PLUS` voneinander getrennt. *Abbildung 7.17* zeigt, welche syntaktischen Möglichkeiten für die Definition eines Sequenzausschnittsteils zur Verfügung stehen.

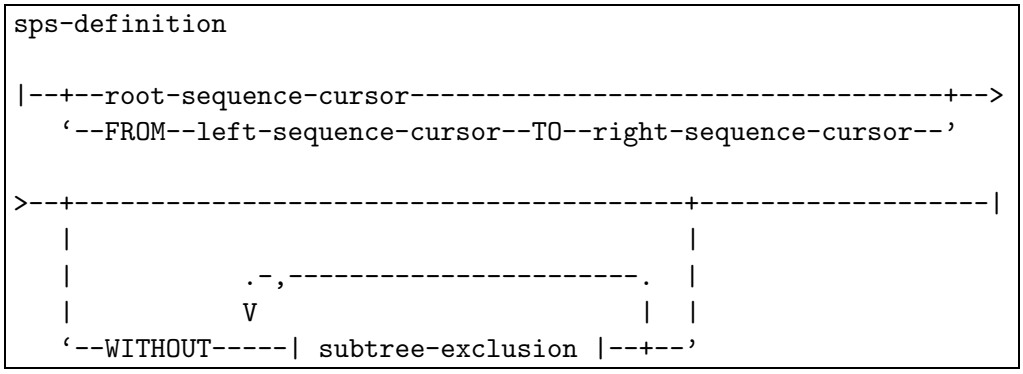


Abbildung 7.17: Syntaxdiagramm für die Definition eines Sequenzausschnittsteils

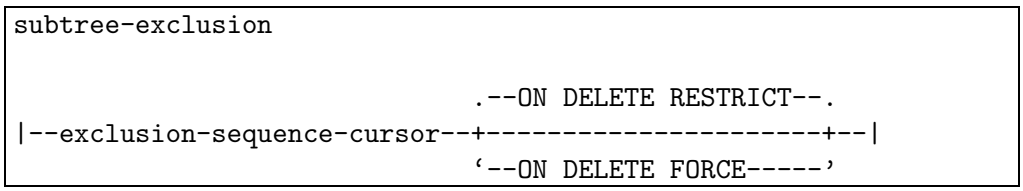


Abbildung 7.18: Syntaxdiagramm für den Ausschluss eines Teilbaums

`root-sequence-cursor`
: Dieser Parameter wird genutzt, wenn es sich beim festzulegenden Sequenzausschnittsteil um einen einzelnen Baum handelt. Mit Hilfe des Parameters `root-sequence-cursor` wird der Name desjenigen Sequenzcursors übergeben, der auf der Wurzel des entsprechenden Baums positioniert ist.

`FROM left-sequence-cursor TO right-sequence-cursor`
: Diese Option wird verwendet, wenn der zu definierende Sequenzausschnittsteil aus mehreren direkt aufeinanderfolgenden Bäumen besteht. Der Parameter `left-sequence-cursor` übergibt dabei den Namen des Sequenzcursors, der auf dem Wurzelknoten des linkesten zum Sequenzausschnittsteil gehörenden Baums steht. Der Parameter `right-sequence-cursor` dient als Platzhalter für den Namen des Sequenzcursors, der die rechte Begrenzung des Sequenzausschnittsteils markiert.

`WITHOUT`
: Bei der Definition eines Sequenzausschnittsteils ist es mit Hilfe des optionalen Schlüsselworts `WITHOUT` möglich, einen oder mehrere Teilbäume auszuschließen. In *Abbildung 7.18* ist dargestellt, welche syntaktischen Möglichkeiten es für den Ausschluss eines Teilbaums gibt.

  `exclusion-sequence-cursor`
  : Mit Hilfe dieses Parameters wird der Name desjenigen Sequenzcursors übergeben, der auf dem Wurzelknoten des auszuschließenden Teilbaums steht.

  `ON DELETE RESTRICT`, `ON DELETE FORCE`
  : Mittels dieser Optionen kann Einfluss auf das Verhalten beim späteren Einbringen der lokalen Änderungen genommen werden. Nähere Einzelheiten hierzu finden sich im Abschnitt 6.3.1.

`ASSIGNING xml-schema`

Mit Hilfe dieser Option wird ein XML-Schema an den zu definierenden Sequenzausschnitt gebunden. Beim Einbringen der lokalen Änderungen kann dieses XML-Schema dann für eine Validitätsprüfung genutzt werden.

Der Platzhalter `xml-schema` repräsentiert die Auswahl eines konkreten XML-Schemas. Für eine solche Schemaauswahl sind dieselben syntaktischen Möglichkeiten nutzbar wie bei der SQL/XML:2006-Funktion `XMLVALIDATE`. Für nähere Details verweisen wir auf [ISO06].

Beispiel

```
DEFINE SEQUENCE PART AS
FROM meinSC1 TO meinSC2 WITHOUT meinSC3 ON DELETE RESTRICT PLUS
meinSC4 WITHOUT meinSC5 ON DELETE FORCE, meinSC6 PLUS
FROM meinSC7 TO meinSC8
ASSIGNING XMLSCHEMA ID unserSchema;
```

Abbildung 7.19: Beispiel für die Anweisung `DEFINE SEQUENCE PART`

Durch die in *Abbildung 7.19* gezeigte Beispielanweisung wird ein aus drei Sequenzausschnittsteilen bestehender Sequenzausschnitt definiert. Dem Sequenzausschnitt wird dabei das XML-Schema *unserSchema* zugeordnet.

Der Sequenzausschnittsteil, der innerhalb der Beispielanweisung zuerst definiert wird, besteht aus mehreren direkt aufeinanderfolgenden Bäumen. Die linke bzw. rechte Begrenzung dieses Sequenzausschnittsteils ergibt sich durch die Positionen der Sequenzcursor *meinSC1* und *meinSC2*. Der Teilbaum, auf dessen Wurzelknoten der Sequenzcursor *meinSC3* positioniert ist, gehört nicht zum Sequenzausschnittsteil. Der Ausschluss dieses Teilbaums erfolgt unter Nutzung der (im Abschnitt 6.3.1 beschriebenen) Option `ON DELETE RESTRICT`.

Der innerhalb unserer Beispielanweisung als zweites definierte Sequenzausschnittsteil ist ein einzelner Baum, auf dessen Wurzelknoten der Sequenzcursor *meinSC4* steht. Der durch den Sequenzcursor *meinSC5* identifizierte Teilbaum (also der Teilbaum, auf dessen Wurzelknoten der Sequenzcursor *meinSC5* positioniert ist) ist kein Bestandteil des Sequenzausschnittsteils. Dieser Teilbaum wird unter Verwendung der Option `ON DELETE FORCE` ausgeschlossen. Auch der Teilbaum, der durch den Sequenzcursor *meinSC6* identifiziert wird, gehört nicht zum Sequenzausschnittsteil. Beim Ausschluss dieses Teilbaums wird (implizit) die Default-Variante `ON DELETE RESTRICT` genutzt.

Der zuletzt definierte Sequenzausschnittsteil umfasst mehrere direkt aufeinanderfolgende Bäume, ohne dass dabei Teilbäume ausgeschlossen sind. Die Sequenzcursor *meinSC7* und *meinSC8* markieren die linke bzw. rechte Begrenzung dieses Sequenzausschnittsteils.

7.3.2 `UNDEFINE SEQUENCE PART`

Die Anweisung `UNDEFINE SEQUENCE PART` *(Abbildung 7.20)* dient zum expliziten Löschen eines Sequenzausschnitts.[5]

[5] Unter dem Löschen eines Sequenzausschnitts verstehen wir nicht das Löschen der im Sequenzausschnitt enthaltenen Knoten, sondern das Löschen der Sequenzausschnittsdefinition (Abschnitt 6.1.1).

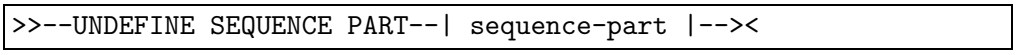


Abbildung 7.20: Syntaxdiagramm der Anweisung `UNDEFINE SEQUENCE PART`

Die Bezugnahme auf den zu löschenden Sequenzausschnitt erfolgt entweder über die Kombination aus Ergebnisspalten*name* und Tupelcursorname oder über die Kombination aus Ergebnisspalten*nummer* und Tupelcursorname *(Abbildung 7.21)*.[6]

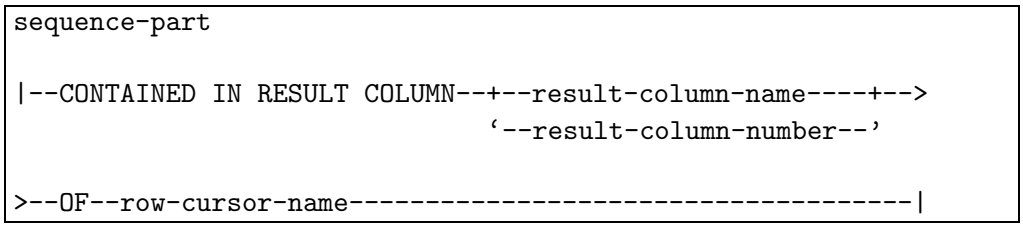


Abbildung 7.21: Syntaxdiagramm für die Bezugnahme auf einen Sequenzausschnitt

Beschreibung der Optionen und Parameter

`result-column-name`
: Dieser Parameter übergibt den Namen der Ergebnisspalte, die den Sequenzausschnitt enthält. Im Anfrageergebnis muss genau eine XML-Spalte mit diesem Namen existieren.

`result-column-number`
: Dieser Platzhalter steht für die (Spalten-)Nummer der Ergebnisspalte, in welcher der Sequenzausschnitt enthalten ist. Es muss eine Ergebnisspalte mit der entsprechenden Nummer geben. Bei dieser Spalte muss es sich um eine XML-Spalte handeln.

`row-cursor-name`
: Mittels dieses Parameters wird der Name des Tupelcursors übergeben, der demjenigen Anfrageergebnis zugeordnet ist, welches den Sequenzausschnitt enthält. Es muss ein geöffneter Tupelcursor mit dem entsprechenden Namen existieren. Dieser Tupelcursor muss auf einem Ergebnistupel positioniert sein.

Beispiel

```
UNDEFINE SEQUENCE PART
CONTAINED IN RESULT COLUMN 21
OF unserTC;
```

Abbildung 7.22: Beispiel für die Anweisung `UNDEFINE SEQUENCE PART`

Mittels der Beispielanweisung *(Abbildung 7.22)* wird derjenige Sequenzausschnitt (explizit) gelöscht, der in der 21. Spalte des mit Hilfe des Tupelcursors *unserTC* erstellten Anfrageergebnisses enthalten ist.

[6] Bei den in den Abschnitten 7.3.3 bis 7.3.6 vorgestellten Sequenzcursoranweisungen erfolgt die Bezugnahme auf den jeweils betroffenen Sequenzausschnitt ebenfalls auf die hier beschriebene Weise.

7.3.3 DESCRIBE SEQUENCE PART

Die Anweisung `DESCRIBE SEQUENCE PART` *(Abbildung 7.23)* wird genutzt, um zu ermitteln, wie groß ein lokaler Speicherbereich mindestens sein muss, damit er einen bestimmten Sequenzausschnitt komplett aufnehmen kann. Die entsprechende Größenangabe wird in der SQLSPCA (Abschnitt 7.4.2) vermerkt.

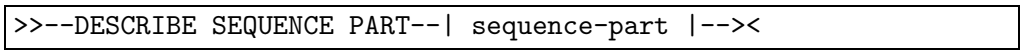


Abbildung 7.23: Syntaxdiagramm der Anweisung `DESCRIBE SEQUENCE PART`

Beschreibung der Optionen und Parameter

`sequence-part`
: Dieser Platzhalter verweist auf das in Abbildung 7.21 dargestellte Syntaxdiagramm. Diesem Syntaxdiagramm kann entnommen werden, welche syntaktischen Möglichkeiten es für die Bezugnahme auf einen konkreten Sequenzausschnitt gibt. Eine Beschreibung der in diesem Syntaxdiagramm enthaltenen Optionen und Parameter findet sich im Abschnitt 7.3.2.

Beispiel

```
DESCRIBE SEQUENCE PART
CONTAINED IN RESULT COLUMN 21
OF unserTC;
```

Abbildung 7.24: Beispiel für die Anweisung `DESCRIBE SEQUENCE PART`

Mit Hilfe der Beispielanweisung *(Abbildung 7.24)* wird ermittelt, wie groß ein lokaler Speicherbereich sein muss, um denjenigen Sequenzausschnitt komplett aufnehmen zu können, der in der 21. Spalte des durch den Tupelcursor *unserTC* identifizierten Anfrageergebnisses enthalten ist.

7.3.4 TRANSFER SEQUENCE PART

Mittels der Anweisung `TRANSFER SEQUENCE PART` *(Abbildung 7.25)* wird ein Sequenzausschnitt ins Anwendungsprogramm übertragen. Die Adresse des zur Aufnahme des Sequenzausschnitts vorgesehenen lokalen Speicherbereichs wird dabei aus der SQLSPCA ausgelesen.

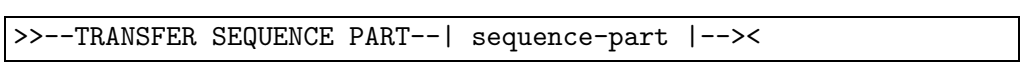


Abbildung 7.25: Syntaxdiagramm der Anweisung `TRANSFER SEQUENCE PART`

Beschreibung der Optionen und Parameter

`sequence-part`
Dieser Platzhalter dient (erneut) als Verweis auf das Syntaxdiagramm aus Abbildung 7.21.

Beispiel

```
TRANSFER SEQUENCE PART
CONTAINED IN RESULT COLUMN 21
OF unserTC;
```

Abbildung 7.26: Beispiel für die Anweisung `TRANSFER SEQUENCE PART`

Die Ausführung der in *Abbildung 7.26* dargestellten Beispielanweisung bewirkt, dass der Sequenzausschnitt, der in der 21. Spalte des mit Hilfe des Tupelcursors *unserTC* erstellten Anfrageergebnisses enthalten ist, ins Anwendungsprogramm übertragen wird.

7.3.5 BRING IN LOCAL CHANGES

Die Anweisung `BRING IN LOCAL CHANGES` *(Abbildung 7.27)* dient dazu, die lokal an einem Sequenzausschnitt vorgenommenen Änderungen in die Datenbank einzubringen. Beim Ausführen dieser Anweisung wird lesend auf die SQLSPCA zugegriffen, um sowohl die Adresse als auch die Größe des lokalen Speicherbereichs zu ermitteln, in dem die (lokal bereitgestellten) Änderungsinformationen enthalten sind.

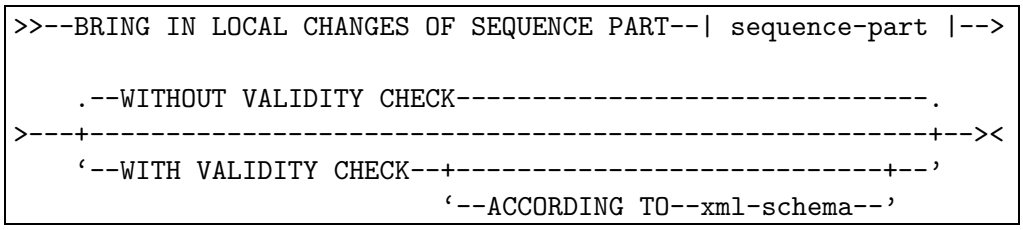


Abbildung 7.27: Syntaxdiagramm der Anweisung `BRING IN LOCAL CHANGES`

Beschreibung der Optionen und Parameter

`sequence-part`
Dieser Platzhalter ist eine Referenz auf das Syntaxdiagramm aus Abbildung 7.21.

`WITHOUT VALIDITY CHECK`
Diese Option ist zu nutzen, wenn beim Einbringen der lokalen Änderungen *keine* Validitätsprüfung vorgenommen werden soll.

`WITH VALIDITY CHECK`
Das Spezifizieren dieser Option bewirkt, dass beim Einbringen der lokalen Änderungen eine Gültigkeitsprüfung durchgeführt wird. Für nähere Details verweisen wir auf Abschnitt 6.6.3.

`ACCORDING TO xml-schema`
: Mit Hilfe dieser Option wird festgelegt, welches XML-Schema bei der Gültigkeitsprüfung zu verwenden ist. Der Platzhalter `xml-schema` repräsentiert dabei (wie auch im Abschnitt 7.3.1) die Auswahl eines konkreten XML-Schemas. Für eine Schemaauswahl stehen (auch hier) dieselben Möglichkeiten zur Verfügung wie bei der SQL/XML:2006-Funktion `XMLVALIDATE`.

  Sofern bereits beim Definieren des Sequenzausschnitts ein XML-Schema festgelegt wurde, kann auf die Nutzung der hier vorgestellten Option verzichtet werden. Die Gültigkeitsprüfung erfolgt dann bezüglich des XML-Schemas, das an den Sequenzausschnitt gebunden ist.

Beispiel

```
BRING IN LOCAL CHANGES OF SEQUENCE PART
CONTAINED IN RESULT COLUMN 21
OF unserTC
WITH VALIDITY CHECK ACCORDING TO XMLSCHEMA ID unserSchema;
```

Abbildung 7.28: Beispiel für die Anweisung `BRING IN LOCAL CHANGES`

Mittels der Beispielanweisung *(Abbildung 7.28)* werden lokale Änderungen in die Datenbank eingebracht. Das Änderungseinbringen bezieht sich dabei auf den Sequenzausschnitt, der in der 21. Spalte des durch den Tupelcursor *unserTC* identifizierten Anfrageergebnisses enthalten ist. Beim Einbringen der lokalen Änderungen wird eine Gültigkeitsprüfung bezüglich des XML-Schemas *unserSchema* vorgenommen.

7.3.6 DISCARD LOCAL CHANGES

Mit Hilfe der Anweisung `DISCARD LOCAL CHANGES` *(Abbildung 7.29)* werden die lokal an einem Sequenzausschnitt vorgenommenen Änderungen explizit verworfen.

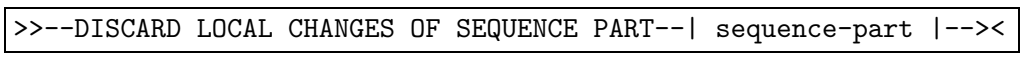


Abbildung 7.29: Syntaxdiagramm der Anweisung `DISCARD LOCAL CHANGES`

Beschreibung der Optionen und Parameter

`sequence-part`
: Dieser Platzhalter dient als Referenz auf das in Abbildung 7.21 dargestellte Syntaxdiagramm.

Beispiel

Durch die Ausführung der Beispielanweisung *(Abbildung 7.30)* werden die lokalen Änderungen desjenigen Sequenzausschnitts verworfen, der in der 21. Spalte des mit Hilfe des Tupelcursors *unserTC* erstellten Anfrageergebnisses enthalten ist.

```
DISCARD LOCAL CHANGES OF SEQUENCE PART
CONTAINED IN RESULT COLUMN 21
OF unserTC;
```

Abbildung 7.30: Beispiel für die Anweisung `DISCARD LOCAL CHANGES`

7.4 Zur Kommunikation dienende Datenstrukturen

In den Abschnitten 7.4.1 und 7.4.2 werden wir näher auf die zur Kommunikation zwischen Anwendungsprogramm und SQL-Laufzeitsystem dienenden Datenstrukturen SQLSCFA und SQLSPCA eingehen. Diese beiden Datenstrukturen wurden von uns zur Unterstützung der Sequenzcursor-basierten Verarbeitung eingeführt.

7.4.1 SQLSCFA (SQL Sequence Cursor Feedback Area)

Die SQLSCFA (SQL Sequence Cursor Feedback Area) ist eine Datenstruktur im Anwendungsprogramm, die genutzt werden kann, um Informationen über Sequenzcursor abzufragen (vgl. Abschnitt 5.3.1). Die SQLSCFA [Jat07, Böh07] wird sowohl nach jeder erfolgreichen Ausführung der Anweisung `MOVE SEQUENCE CURSOR` (Abschnitt 7.2.3), als auch nach jeder erfolgreichen Ausführung der Anweisung `GET INFORMATION ABOUT SEQUENCE CURSOR` (Abschnitt 7.2.4) automatisch vom SQL-Laufzeitsystem aktualisiert. Die dabei in der SQLSCFA vermerkten Informationen beziehen sich auf denjenigen Sequenzcursor, auf den in der Anweisung `MOVE SEQUENCE CURSOR` bzw. `GET INFORMATION ABOUT SEQUENCE CURSOR` Bezug genommen wurde. Das Anwendungsprogramm kann lesend auf die SQLSCFA zugreifen.

Die SQLSCFA setzt sich aus elf Komponenten zusammen. Im Abschnitt 7.4.1.1 beschreiben wir diese Komponenten zunächst für den Fall des *nicht*-erweiterten Sequenzcursor-basierten Verarbeitungsablaufs. Dabei werden wir für jede SQLSCFA-Komponente jeweils auch einen geeigneten Datentyp angeben. Der Abschnitt 7.4.1.2 befasst sich anschließend mit den Änderungen, die sich ergeben, wenn der *erweiterte* Verarbeitungsablauf zugrunde gelegt wird.

7.4.1.1 Beschreibung der SQLSCFA-Komponenten (nicht-erweiterter Ablauf)

Die SQLSCFA besteht aus den im Folgenden aufgeführten Komponenten. Bei der Beschreibung dieser Komponenten setzen wir den nicht-erweiterten Sequenzcursor-basierten Verarbeitungsablauf voraus.

sequence_cursor_state `CHAR(1)`
: In dieser Komponente ist der Status des entsprechenden Sequenzcursors verzeichnet. Dabei sind die beiden folgenden Werte möglich:
  'P' — positioned („positioniert“)
  'N' — not positioned („unpositioniert“)
  Falls diese Komponente den Wert *'N'* enthält, sind die Inhalte der (im Folgenden beschriebenen) SQLSCFA-Komponenten *node_kind*, *node_type*, *node_type_truncated*,

node_name, *node_name_truncated*, *position_info* und *position_info_truncated* zu ignorieren.

node_kind `CHAR(1)`
Sofern die SQLSCFA-Komponente *sequence_cursor_state* nicht den Wert *'N'* besitzt, gibt die Komponente *node_kind* Auskunft über die Art desjenigen Knotens, auf dem der Sequenzcursor positioniert ist. Es können folgende sieben Werte auftreten:
'D' — Document Node (D-Knoten)
'E' — Element Node (E-Knoten)
'A' — Attribute Node (A-Knoten)
'P' — Processing Instruction Node (P-Knoten)
'C' — Comment Node (C-Knoten)
'T' — Text Node (T-Knoten)
'V' — Atomic Value Node (V-Knoten)

node_type `VARCHAR(128)`
Der Wert dieser Komponente ist zu ignorieren, falls die Komponente *sequence_cursor_state* den Wert *'N'* besitzt und/oder die Komponente *node_kind* einen der Werte *'D'*, *'P'*, *'C'* oder *'T'* enthält. Andernfalls kann der Komponente *node_type* der Typ des Knotens entnommen werden, auf dem der Sequenzcursor steht. Falls die Typbezeichnung des Knotens jedoch länger als 128 Zeichen ist, werden lediglich die ersten 128 Zeichen geliefert.

Anstatt die in der SQLSCFA ablegbaren Typbezeichnungen (wie von uns vorgesehen) auf 128 Zeichen zu beschränken, wäre natürlich auch eine Längenbeschränkung auf eine andere Anzahl von Zeichen denkbar. Ein Zulassen längerer Typbezeichnungen hätte den Vorteil, dass es dann seltener notwendig wäre, eine Kürzung der Typbezeichnung durchzuführen. Dem stände allerdings der Nachteil entgegen, dass die SQLSCFA mehr Speicherplatz benötigen würde, wodurch das (häufig notwendig werdende) Aktualisieren dieser Datenstruktur mit mehr Aufwand verbunden wäre.

Eine Längenbeschränkung auf 128 Zeichen stellt unserer Ansicht nach einen „vernünftigen Kompromiss“ dar: Trotz einer vertretbaren Größe der SQLSCFA-Komponente *node_type* kann in vielen (für die Praxis relevanten) Fällen auf eine Kürzung des Typnamens verzichtet werden.[7] Wir haben auch für einige der anderen SQLSCFA-Komponenten eine Längenbeschränkung auf 128 Zeichen vorgesehen. Die zuvor durchgeführten Betrachtungen gelten analog auch für diese Komponenten.

node_type_truncated `CHAR(1)`
Diese Komponente gibt an, ob die Komponente *node_type* anstatt des kompletten Typnamens nur eine *gekürzte* Fassung der Typbezeichnung enthält:
'Y' — Yes (Die Typbezeichnung wurde gekürzt.)
'N' — No (Es wurde keine Kürzung der Typbezeichnung vorgenommen.)

node_name `VARCHAR(128)`
Der Wert dieser Komponente ist zu ignorieren, falls die Komponente *sequence_cursor_state* den Wert *'N'* enthält und/oder die Komponente *node_kind* einen der Werte *'D'*, *'C'*, *'T'* oder *'V'* besitzt. In allen anderen Fällen entspricht

[7] In mehreren vergleichbaren Situationen sehen die SQL-Norm [ISO03c] bzw. DB2 [IBM07b] ebenfalls eine Längenbeschränkung auf 128 Zeichen vor.

der Inhalt der Komponente *node_name* dem Namen des Knotens, auf dem der Sequenzcursor positioniert ist. Sollte der Knotenname jedoch länger als 128 Zeichen sein, werden nur die ersten 128 Zeichen berücksichtigt.

node_name_truncated `CHAR(1)`
Dieser Komponente kann entnommen werden, ob die Komponente *node_name* eine *gekürzte* Fassung des Knotennamens enthält:
'Y' — Yes (Es wurde eine Kürzung des Knotennamens vorgenommen.)
'N' — No (Der Knotenname wurde nicht gekürzt.)

position_info `VARCHAR(128)`
Sofern die Komponente *sequence_cursor_state* nicht den Wert *'N'* besitzt, enthält die Komponente *position_info* eine auf den Sequenzcursor bezogene *Positionsinformation*. Bei dieser Positionsinformation handelt es sich um die (gegebenenfalls auf die ersten 128 Zeichen gekürzte) *Positionsnummer* des Knotens, auf dem der Sequenzcursor steht. Eine solche Positionsnummer kann aus mehreren Teilen bestehen. Die einzelnen Teile einer mehrteiligen Positionsnummer sind durch einen Punkt (.) voneinander getrennt.

Die Positionsnummern der Knoten, die in einer (in typed-value-orientierter Repräsentation vorliegenden) XQuery-Sequenz enthalten sind, ergeben sich eindeutig gemäß der folgenden drei Regeln:

1. Der Wurzelknoten des i-ten Baums erhält die (einteilige) Positionsnummer i. Beispielsweise besitzt der Wurzelknoten vom dritten Baum die Positionsnummer *3*, während die Positionsnummer *21* den Wurzelknoten des 21. Baums identifiziert.
2. Dem j-ten Kindknoten des Knotens mit der Positionsnummer $<VaterNr>$ wird die Positionsnummer $<VaterNr>.j$ zugeteilt. Somit besitzt z. B. der zweitlinkeste Kindknoten vom Wurzelknoten des dritten Baums die Positionsnummer *3.2*, während die Positionsnummer *3.2.6* wiederum zu dessen sechstlinkesten Kindknoten gehört.
3. Der k-te A-Knoten, der dem Knoten mit der Positionsnummer $<VaterNr>$ zugeordnet ist, erhält die Positionsnummer $<VaterNr>.–k$. Sind dem Wurzelknoten des 21. Baums beispielsweise fünf A-Knoten zugeordnet, so besitzen diese (von links nach rechts) die Positionsnummern *21.–1*, *21.–2*, *21.–3*, *21.–4* und *21.–5*.

Es wäre prinzipiell natürlich denkbar, anstelle des zuvor beschriebenen Knotennummerierungsverfahrens eine andere Nummerierungsvariante zu nutzen. So könnte beispielsweise eine auf der Präorder-Reihenfolge basierende Nummerierung der Knoten durchgeführt werden (vgl. Abschnitt 6.1.4). Für die von uns vorgesehene Art der Knotennummerierung spricht jedoch der Vorteil, dass die Knotennummern sehr leicht „auswertbar“ sind:

- Dem ersten Teil der Knotennummer kann direkt entnommen werden, zu welchem Baum der entsprechende Knoten gehört. Beispielsweise befindet sich der Knoten mit der Knotennummer *14.9.51.15.1.51* im 14. Baum der Sequenz.

- Die Anzahl der Nummernteile (bzw. die Anzahl der in der Knotennummer enthaltenen Punkte) gibt Auskunft darüber, in welcher Ebene der Sequenz sich der Knoten befindet. So ist z. B. der Knoten mit der einteiligen Knotennummer *5* in der Wurzelebene der Sequenz enthalten, während der Knoten mit der dreiteiligen Knotennummer *21.5.77* zur drittobersten Ebene gehört.
- Aus der Knotennummer eines Knotens sind unkompliziert die Knotennummern aller seiner Vorfahren bestimmbar. Beispielsweise besitzt der Vater vom Knoten mit der Knotennummer *3.2.6* die Knotennummer *3.2*. Dessen Vater hat wiederum die Knotennummer *3*. Somit ist leicht ermittelbar, ob ein Knoten der Vorfahre eines anderen Knotens ist. So ist z. B. der Knoten mit der Knotennummer *18.2.25* ein Vorfahre des Knotens mit der Knotennummer *18.2.25.5.7.76*. Die Knoten mit den Knotennummern *18.2.25* und *27.2.27* stehen hingegen in keiner Vorfahre-Nachfahre-Beziehung.

position_info_truncated `CHAR(1)`
Diese Komponente gibt Auskunft darüber, ob in der Komponente *position_info* eine *gekürzte* Fassung der Positionsinformation enthalten ist:
'Y' — Yes (Es hat eine Kürzung der Positionsinformation stattgefunden.)
'N' — No (Es wurde keine Kürzung der Positionsinformation durchgeführt.)

row_cursor_name `VARCHAR(128)`
In dieser Komponente ist der Name des Tupelcursors vermerkt, der dem Sequenzcursor zugeordnet ist. Falls der Tupelcursorname länger als 128 Zeichen ist, werden nur die ersten 128 Zeichen geliefert.

row_cursor_name_truncated `CHAR(1)`
Diese Komponente gibt an, ob die Komponente *row_cursor_name* eine *gekürzte* Fassung des Tupelcursornamens enthält:
'Y' — Yes (Es wurde eine Kürzung des Tupelcursornamens durchgeführt.)
'N' — No (Es hat keine Kürzung des Tupelcursornamens stattgefunden.)

result_column_number `INTEGER`
Dieser Komponente kann entnommen werden, an die wievielte Spalte des Anfrageergebnisses der Sequenzcursor gebunden ist.

7.4.1.2 Änderungen beim Zugrundelegen des erweiterten Ablaufs

Wird anstelle des nicht-erweiterten Sequenzcursor-basierten Verarbeitungsablaufs der erweiterte Sequenzcursor-basierte Verarbeitungsablauf vorausgesetzt, so kann die SQLSCFA-Komponente *sequence_cursor_state* außer den Werten *'P'* und *'N'* auch die folgenden vier Werte annehmen:[8]
'L' — left of the sequence („links neben der Sequenz“)
'R' — right of the sequence („rechts neben der Sequenz“)
'B' — between two trees („zwischen zwei Bäumen“)
'U' — under a node („unterhalb eines Knotens“)

[8] Die Werte *'P'* und *'N'* besitzen hier die gleiche Bedeutung wie beim nicht-erweiterten Sequenzcursor-basierten Verarbeitungsablauf.

Besitzt die SQLSCFA-Komponente *sequence_cursor_state* den Wert *'N'* (der Sequenzcursor befindet sich also im Status *„unpositioniert“*), so sind die Inhalte der sieben Knoten-basierten SQLSCFA-Komponenten zu ignorieren. Unter den *Knoten-basierten* SQLSCFA-Komponenten verstehen wir dabei die Komponenten *node_kind*, *node_type*, *node_type_truncated*, *node_name*, *node_name_truncated*, *position_info* und *position_info_truncated*.

Falls die SQLSCFA-Komponente *sequence_cursor_state* jedoch einen von *'N'* verschiedenen Wert hat, so gibt dieser Wert Auskunft darüber, auf welchen Knoten sich die Inhalte der Knoten-basierten SQLSCFA-Komponenten beziehen. Nähere Einzelheiten hierzu finden sich in *Abbildung 7.31*.

Status[1]	Knoten, auf den sich die Inhalte der Knoten-basierten SQLSCFA-Komponenten beziehen
'P'	Knoten, auf dem der Sequenzcursor positioniert ist
'L'	Wurzelknoten des linkesten Baums der Sequenz
'R'	Wurzelknoten des rechtesten Baums der Sequenz
'B'	Wurzelknoten des rechteren der beiden benachbarten Bäume, zwischen denen sich der Sequenzcursor befindet
'U'	Knoten, unterhalb dem sich der Sequenzcursor befindet

[1] Inhalt der SQLSCFA-Komponente *sequence_cursor_state*

Abbildung 7.31: Bezugsknoten der Knoten-basierten SQLSCFA-Komponenten

7.4.2 SQLSPCA (SQL Sequence Part Communications Area)

Die SQLSPCA (SQL Sequence Part Communications Area) ist eine Datenstruktur im Anwendungsprogramm, auf die beim Ausführen der Anweisungen `DESCRIBE SEQUENCE PART` (Abschnitt 7.3.3), `TRANSFER SEQUENCE PART` (Abschnitt 7.3.4) und `BRING IN LOCAL CHANGES` (Abschnitt 7.3.5) zugegriffen wird. Die SQLSPCA [Jat07, Böh07, Ott07] besteht aus den folgenden beiden Komponenten:

size_of_required_storage `INTEGER`

Der Inhalt dieser Komponente ist ein Integer-Wert, der eine Größenangabe repräsentiert. Die (implizite) Maßeinheit dieser Größenangabe ist *Byte*.

Beim Ausführen der Anweisung `DESCRIBE SEQUENCE PART` vermerkt das SQL-Laufzeitsystem in dieser Komponente, wie groß ein lokaler Speicherbereich mindestens sein muss, damit er den kompletten Sequenzausschnitt aufnehmen kann. Diese Größenangabe wird anschließend vom Anwendungsprogramm ausgelesen und genutzt (Abschnitt 4.2.6).

Vor der Ausführung der Anweisung `BRING IN LOCAL CHANGES` vermerkt das Anwendungsprogramm in dieser Komponente, wie groß der lokale Speicherbereich ist, der die lokal bereitgestellten Änderungsinformationen enthält. Beim Einbringen der lokalen Änderungen wird vom SQL-Laufzeitsystem dann auf diese Größenangabe zugegriffen (Abschnitt 4.2.8.1).

pointer_to_local_storage `POINTER`

Beim Inhalt dieser Komponente handelt es sich um die (Anfangs-)Adresse eines lokalen Speicherbereichs.

Vor dem Ausführen der Anweisung `TRANSFER SEQUENCE PART` wird vom Anwendungsprogramm in dieser Komponente die Adresse des Speicherbereichs vermerkt, der den Sequenzausschnitt lokal aufnehmen soll. Beim Übertragen des Sequenzausschnitts greift das SQL-Laufzeitsystem dann auf diese Adressangabe zu (Abschnitt 4.2.6).

Bevor die Anweisung `BRING IN LOCAL CHANGES` ausgeführt wird, vermerkt das Anwendungsprogramm in dieser Komponente die Adresse des lokalen Speicherbereichs, der die Änderungsinformationen enthält. Auf diese Adressangabe wird vom SQL-Laufzeitsystem dann bei der Anweisungsausführung zugegriffen (Abschnitt 4.2.8.1).

Abbildung 7.32 fasst die Nutzung der beiden SQLSPCA-Komponenten nochmals kompakt zusammen.

Sequenzcursoranweisung	*size_of_required_storage*	*pointer_to_local_storage*
`DESCRIBE SEQUENCE PART`	Größe, die ein lokaler Speicherbereich mindestens haben muss, um den Sequenzausschnitt komplett aufnehmen zu können	*(ungenutzt)*
`TRANSFER SEQUENCE PART`	*(ungenutzt)*	Adresse des lokalen Speicherbereichs, der den Sequenzausschnitt aufnehmen soll
`BRING IN LOCAL CHANGES`	Größe des lokalen Speicherbereichs, der die Änderungsinformationen enthält	Adresse des lokalen Speicherbereichs, der die Änderungsinformationen enthält

Abbildung 7.32: Nutzung der SQLSPCA-Komponenten

Kapitel 8

Prototypische Realisierung

Im Rahmen der vorliegenden Arbeit und zuarbeitender Diplomarbeiten [Kie06, Jat07, Böh07, Ott07] wurden die wesentlichen Konzepte der Sequenzcursor-basierten Verarbeitung prototypisch implementiert [MO08]. Das dabei verfolgte Hauptziel bestand darin, die Realisierbarkeit der von uns eingeführten Konzepte zu zeigen. Darüber hinaus ging es bei der Prototypentwicklung darum, eine Testplattform bereitzustellen, mit deren Hilfe sich praktische Erfahrungen im Umgang mit den Sequenzcursoranweisungen und dem SQL/XML:2006-Basisdatentyp `XML` sammeln lassen. Performance-Aspekte spielten bei der Erstellung des Prototyps hingegen nur eine untergeordnete Rolle.

Im aktuellen Kapitel gehen wir näher auf die prototypische Realisierung des Sequenzcursor-basierten Verarbeitungsmodells ein. Nach der Durchführung von Architekturbetrachtungen im Abschnitt 8.1 beschreiben wir im Abschnitt 8.2 die Implementierung des von uns entwickelten Prototyps. Abschnitt 8.3 beschließt das Kapitel mit einem kurzen Fazit.

8.1 Architekturbetrachtungen

Im Folgenden werden wir die Architektur des Prototyps Schritt für Schritt entwickeln. Als Ausgangspunkt dient uns dabei die Architektur, die vorliegen würde, wenn die Sequenzcursor-basierte Verarbeitungsfunktionalität vollständig ins DBMS integriert wäre. Diese Architektur wird im Abschnitt 8.1.1 vorgestellt. Bei den anschließend in den Abschnitten 8.1.2 und 8.1.3 betrachteten Architekturvarianten handelt es sich lediglich um „Zwischenergebnisse“, die bei der schrittweisen Entwicklung der Prototyparchitektur auftreten. Abschnitt 8.1.4 beschreibt schließlich die endgültige Prototyparchitektur — also die Architektur des tatsächlich implementierten Prototyps.

8.1.1 Architektur bei vollständiger Integration

Die Sequenzcursoranweisungen wurden von uns als (mögliche) Erweiterung der Datenbanksprache SQL konzipiert. Die entsprechend erweiterte Fassung von SQL bezeichnen wir im Folgenden als *SQL$^+$*. Unter einem *SQL$^+$-DBMS* verstehen wir ein Datenbankmanagementsystem, das SQL$^+$ vollständig unterstützt.

Abbildung 8.1 zeigt die Architektur, die sich beim Einsatz eines SQL$^+$-DBMS ergeben würde. Diese Architektur besteht aus zwei Schichten:

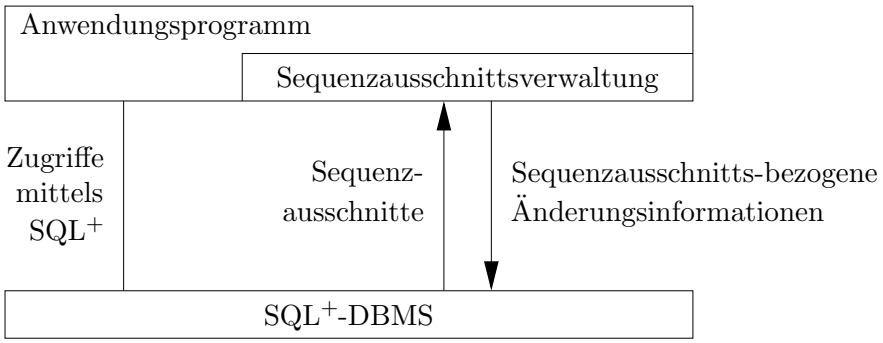


Abbildung 8.1: Architektur bei vollständiger Integration

SQL$^+$-DBMS
: Das SQL$^+$-DBMS behandelt sämtliche Sequenzcursoranweisungen wie „normale“ SQL-Anweisungen. Bei der Anweisungsverarbeitung wird konzeptuell also nicht zwischen Sequenzcursoranweisungen und traditionellen SQL-Anweisungen unterschieden — die Sequenzcursor-basierte Verarbeitungsfunktionalität ist vollständig ins DBMS (d. h. in den DBMS-Kern) integriert. Das Anwendungsprogramm kann folglich mit beliebigen SQL$^+$-Anweisungen — also sowohl mit Sequenzcursoranweisungen als auch mit herkömmlichen SQL-Anweisungen — auf das DBMS zugreifen.

Anwendungsprogramm mit Sequenzausschnittsverwaltung
: Das Anwendungsprogramm definiert Sequenzausschnitte, um diese lokal verarbeiten zu können. Die definierten Sequenzausschnitte werden anschließend vom SQL$^+$-DBMS ins Anwendungsprogramm übertragen und dort lokal von der *Sequenzausschnittsverwaltung* verwaltet. Die Sequenzausschnittsverwaltung, bei der es sich aus Architektursicht um einen Teil des Anwendungsprogramms handelt, ermöglicht ein lokales Arbeiten auf den Sequenzausschnitten und protokolliert die dabei vorgenommenen Änderungen [Ott07].

  Auf Anforderung stellt die Sequenzausschnittsverwaltung sämtliche Änderungsinformationen, die einen bestimmten Sequenzausschnitt betreffen, in einem zusammenhängenden lokalen Speicherbereich bereit. Diese Sequenzausschnitts-bezogenen Änderungsinformationen werden dann vom SQL$^+$-DBMS ausgelesen und zum Einbringen der lokalen Änderungen genutzt.

Die Implementierung eines voll funktionsfähigen SQL$^+$-DBMS ’from scratch’ kommt für uns aus Aufwandsgründen nicht in Frage. Für eine prototypische Realisierung ist es stattdessen sinnvoll, ein SQL/XML:2006-DBMS — also ein SQL/XML:2006-fähiges DBMS-Produkt — als Ausgangsbasis zu nutzen und nur die zusätzliche Funktionalität zu ergänzen. Da uns eine „echte“ Integration der Zusatzfunktionalität ins SQL/XML:2006-DBMS (d. h. in den DBMS-Kern) nicht möglich ist (eine umfangreiche Änderung des komplexen DBMS-Quellcodes ist im Rahmen der vorliegenden Arbeit nicht realisierbar), bietet sich die Verwendung einer Schichtenarchitektur an. Die neu eingeführte Funktionalität wird dabei mit Hilfe einer oder mehrerer zusätzlicher Schichten auf ein vorhandenes SQL/XML:2006-DBMS aufgesetzt. Wir werden im folgenden Abschnitt näher auf diesen Ansatz eingehen.

8.1.2 Prototyparchitektur — Variante 1

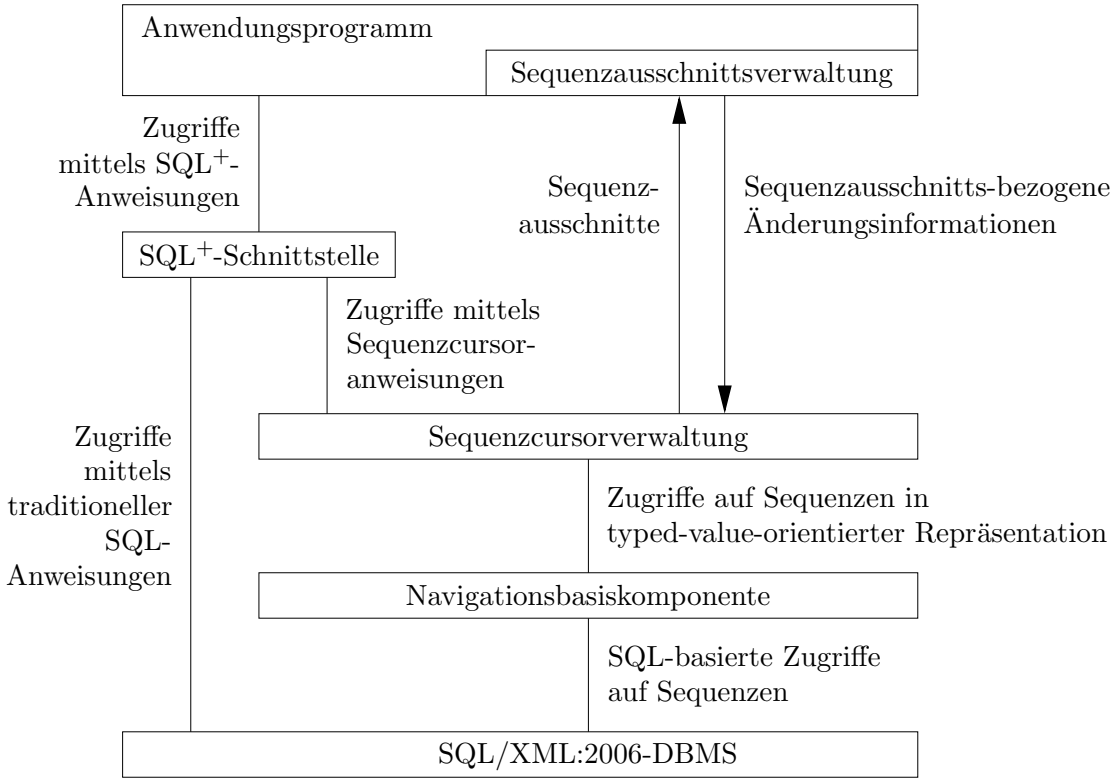


Abbildung 8.2: Prototyparchitektur — Variante 1

Bei der in *Abbildung 8.2* dargestellten Architektur wird die Sequenzcursor-basierte Verarbeitungsfunktionalität durch mehrere zusätzliche Schichten realisiert, die auf ein SQL/XML:2006-DBMS aufgesetzt sind. Insgesamt besteht die Architektur aus fünf Schichten:

SQL/XML:2006-DBMS
: Das SQL/XML:2006-DBMS ist ein (fiktives) Datenbankmanagementsystem, welches SQL/XML:2006 sowie die anderen Teile der aktuellen SQL-Norm unterstützt. Wir setzen hierbei voraus, dass die XML-Unterstützung nicht über die Vorgaben der SQL-Norm hinausgeht.

  Die einzige vom SQL/XML:2006-DBMS angebotene Möglichkeit, lesend auf eine im aktuellen Ergebnistupel enthaltene Sequenz zuzugreifen, besteht darin, die Sequenz serialisiert auszulesen. Änderungen an einer im aktuellen Ergebnistupel enthaltenen Sequenz (bzw. an der ihr zugrunde liegenden Ursprungssequenz der Basistabelle) können mit Hilfe eines positionierten `UPDATE`s realisiert werden. Auf der rechten Seite der zur `UPDATE`-Anweisung gehörenden `SET`-Klausel muss dabei ein Ausdruck spezifiziert werden, dessen Wert der geänderten Sequenz entspricht.

Navigationsbasiskomponente
: Diese Komponente ermöglicht es der (die nächst höhere Schicht darstellenden) *Sequenzcursorverwaltung*, auf die im aktuellen Ergebnistupel enthaltenen Sequenzen

zuzugreifen, als würden diese Sequenzen in typed-value-orientierter Repräsentation vorliegen [Jat07]. Da diese Art der Sequenzrepräsentation als Basis für die Navigation der Sequenzcursor dient, wird die hier beschriebene Komponente als *Navigationsbasiskomponente* bezeichnet.

Mit Hilfe der vom SQL/XML:2006-DBMS angebotenen Möglichkeiten liest die Navigationsbasiskomponente die im aktuellen Ergebnistupel enthaltenen Sequenzen serialisiert aus. Die als Zeichenketten vorliegenden Sequenzen werden von der Navigationsbasiskomponente anschließend geparst und gegebenenfalls einer Validierung (einschließlich der damit verbundenen Anreicherung mit Typinformationen) unterzogen. Danach überführt die Navigationsbasiskomponente die Sequenzen in die typed-value-orientierte Repräsentation.

Der Sequenzcursorverwaltung werden Zugriffsmöglichkeiten bereitgestellt, die einen lesenden Zugriff auf die in typed-value-orientierter Repräsentation vorliegenden Sequenzen gestatten. Außerdem wird es der Sequenzcursorverwaltung ermöglicht, Änderungen in diese Sequenzen (bzw. in die diesen Sequenzen zugrunde liegenden Ursprungssequenzen) einzubringen.

Um Änderungen in eine Sequenz einzubringen, übergibt die Sequenzcursorverwaltung die auf diese Sequenz bezogenen Änderungsinformationen an die Navigationsbasiskomponente. Die Navigationsbasiskomponente arbeitet diese Änderungen zunächst in die entsprechende (in typed-value-orientierter Repräsentation vorliegende) Sequenz ein und führt anschließend eine Rücktransformation[1] der geänderten Sequenz durch. Basierend auf der geänderten (in traditioneller Sequenzrepräsentation vorliegenden) Sequenz erzeugt die Navigationsbasiskomponente eine positionierte `UPDATE`-Anweisung, bei der die rechte Seite der `SET`-Klausel einen Ausdruck enthält, dessen Wert der geänderten Sequenz entspricht. Diese positionierte `UPDATE`-Anweisung wird dann vom SQL/XML:2006-DBMS verarbeitet, wodurch die Änderungen in die Datenbank eingebracht werden.

Sequenzcursorverwaltung

Diese Komponente nimmt von der *SQL$^+$-Schnittstelle* (bei der es sich um die nächst höhere Schicht handelt) Sequenzcursoranweisungen entgegen und verarbeitet diese. Die Sequenzcursorverwaltung ist außerdem für die Verwaltung der Sequenzausschnittsbeschreibungsdaten[2] und — wie es ihr Name nahelegt — für die Verwaltung der Sequenzcursor zuständig [Böh07].

Die Sequenzcursorverwaltung ermöglicht (u. a.) die Definition und Übertragung von Sequenzausschnitten. Beim „Zusammenbauen“ der zu übertragenden Sequenzausschnitte greift die Sequenzcursorverwaltung lesend auf die typed-value-orientierte Repräsentation der im aktuellen Ergebnistupel enthaltenen Sequenzen zu. Die Sequenzcursorverwaltung nutzt dabei die ihr von der Navigationsbasiskomponente zur Verfügung gestellten Zugriffsmöglichkeiten. Die fertigen Sequenzausschnitte werden dann ins Anwendungsprogramm übertragen, wo sie von der Sequenzausschnittsverwaltung lokal verwaltet werden.

[1] Unter einer *Rücktransformation* verstehen wir die Überführung einer in typed-value-orientierter Repräsentation vorliegenden Sequenz in die traditionelle Sequenzrepräsentation.

[2] Die Sequenzausschnittsbeschreibungsdaten geben beispielsweise Auskunft darüber, aus wie vielen Sequenzausschnittsteilen ein Sequenzausschnitt besteht und welchen SAT-Klassen diese Sequenzausschnittsteile zuzuordnen sind.

Um die lokal an einem Sequenzausschnitt vorgenommenen Änderungen in die Datenbank einzubringen, liest die Sequenzcursorverwaltung die von der Sequenzausschnittsverwaltung lokal bereitgestellten — auf den entsprechenden Sequenzausschnitt bezogenen — Änderungsinformationen aus. Die Sequenzcursorverwaltung überführt diese Sequenzausschnitts-bezogenen Änderungsinformationen anschließend in Sequenz-bezogene Änderungsinformationen. Diese Änderungsinformationen werden dann zur weiteren Verarbeitung an die Navigationsbasiskomponente übergeben.

SQL$^+$-Schnittstelle
Auf diese Komponente wird vom Anwendungsprogramm mit Hilfe von SQL$^+$-Anweisungen zugegriffen. Je nachdem, ob es sich bei einer SQL$^+$-Anweisung um eine Sequenzcursoranweisung oder um eine traditionelle SQL-Anweisung handelt, wird diese Anweisung an die Sequenzcursorverwaltung bzw. ans SQL/XML:2006-DBMS weitergeleitet. Die entsprechende Komponente ist dann für die Ausführung der Anweisung verantwortlich. Etwaige infolge der Anweisungsausführung erzeugte Rückgaben werden zunächst an die SQL$^+$-Schnittstelle (zurück)geliefert und von dieser dann ans Anwendungsprogramm durchgereicht.

Eine andere Aufgabe der SQL$^+$-Schnittstelle besteht darin, ein Auftreten jener traditionellen SQL-Anweisungen zu erkennen, die für die Sequenzcursor-basierte Verarbeitung unmittelbar relevant sind. Zu diesen Anweisungen zählt beispielsweise die zum Weiterbewegen des Tupelcursors genutzte SQL-`FETCH`-Anweisung. Eine Ausführung dieser Anweisung wirkt sich auf die Sequenzcursor-basierte Verarbeitung wie folgt aus:

- Sämtliche Sequenzcursor, denen der entsprechende Tupelcursor zugeordnet ist, werden in den Status *„unpositioniert“* versetzt (Abschnitt 5.4).
- Sofern die lokal am Sequenzausschnitt vorgenommenen Änderungen noch nicht eingebracht (oder explizit verworfen) wurden, findet ein implizites Verwerfen dieser Änderungen statt (Abschnitt 4.2.8.2). Dies gilt für jeden Sequenzausschnitt, der in demjenigen Anfrageergebnis enthalten ist, zu dem der Tupelcursor gehört.
- Alle im entsprechenden Anfrageergebnis enthaltenen Sequenzausschnitte werden implizit gelöscht (Abschnitt 6.1.1).

Die SQL$^+$-Schnittstelle meldet der Sequenzcursorverwaltung jedes Auftreten einer für die Sequenzcursor-basierte Verarbeitung relevanten SQL-Anweisung. Die Sequenzcursorverwaltung kann damit geeignet auf die entsprechende Anweisung reagieren und beispielsweise den Status von Sequenzcursorn anpassen oder Sequenzausschnitte implizit löschen.

Anwendungsprogramm mit Sequenzausschnittsverwaltung
Diese Schicht erfüllt dieselben Aufgaben wie die gleichnamige Schicht der im Abschnitt 8.1.1 betrachteten Architektur. Bezüglich des Zusammenspiels mit anderen Schichten ergeben sich gegenüber unseren Ausführungen aus Abschnitt 8.1.1 jedoch folgende Änderungen:

- Das Anwendungsprogramm greift mit Hilfe der SQL$^+$-Anweisungen auf die SQL$^+$-Schnittstelle (und nicht auf das SQL$^+$-DBMS) zu.

- Die Sequenzausschnitte werden von der Sequenzcursorverwaltung (und nicht vom SQL$^+$-DBMS) ins Anwendungsprogramm übertragen.
- Das Auslesen der Sequenzausschnitts-bezogenen Änderungsinformationen wird von der Sequenzcursorverwaltung (und nicht vom SQL$^+$-DBMS) durchgeführt.

Die hier vorgestellte (auf einem SQL/XML:2006-DBMS basierende) Schichtenarchitektur besitzt jedoch vier wesentliche Nachteile:

1. Die Navigationsbasiskomponente liest die im aktuellen Ergebnistupel enthaltenen Sequenzen *komplett* aus, obwohl für die weitere Verarbeitung i. d. R. nur Ausschnitte dieser Sequenzen benötigt werden.

2. Die mit Typinformationen angereicherten Sequenzen werden beim Auslesen serialisiert, wodurch alle Typinformationen verloren gehen. Anschließend werden die Sequenzen erneut geparst und wieder mit Typinformationen angereichert.

3. Gemäß SQL/XML:2006 ist ein serialisiertes Auslesen nur für Sequenzen möglich, die aus (maximal) einem Sequenzeintrag bestehen (Abschnitt 4.5.1). Die beschriebene Architektur ist somit nicht für Sequenzen nutzbar, die sich aus *mehreren* Sequenzeinträgen zusammensetzen.

4. Die zum Einbringen der lokalen Änderungen von der Navigationsbasiskomponente generierte positionierte `UPDATE`-Anweisung enthält auf der rechten Seite der `SET`-Klausel einen Ausdruck, dessen Wert der geänderten Sequenz entspricht. Aufgrund fehlender Sprachmittel ist es dabei jedoch *nicht* möglich, innerhalb des Ausdrucks auf die alte Sequenz Bezug zu nehmen und lediglich die vorzunehmenden Änderungen zu spezifieren: Mit Hilfe des Ausdrucks muss die aktualisierte Sequenz stattdessen von Grund auf neu konstruiert werden. Ein solches Vorgehen ist insbesondere dann sehr ineffektiv, wenn an großen Sequenzen lediglich kleine Änderungen vorgenommen werden.

Die Ursache aller zuvor aufgeführten Nachteile besteht darin, dass es mit SQL/XML:2006 nicht möglich ist, adäquat auf die im aktuellen Ergebnistupel enthaltenen Sequenzen zuzugreifen. Unserer Ansicht nach sind (vollkommen unabhängig von der von uns vorgeschlagenen Sequenzcursor-basierten Verarbeitung) folgende grundlegende SQL/XML:2006-Erweiterungen dringend notwendig:

1. Eine im aktuellen Ergebnistupel enthaltene Sequenz kann sehr groß sein. Es sollte deshalb möglich sein, lesend auf *Teile* dieser Sequenz (also auf einzelne Sequenzeinträge oder Knoten) zugreifen zu können, ohne dass dazu die gesamte Sequenz ausgelesen werden muss.

   Um dies zu realisieren, wäre ein Vorgehen denkbar, welches an das im Abschnitt 2.5 vorgestellte LOB-Locator-Prinzip angelehnt ist: Ein so genannter *Sequenz-Locator* würde beim `FETCH` nicht die eigentliche Sequenz aufnehmen, sondern nur einen Verweis auf diese. In anderen SQL-Anweisungen könnte der Sequenz-Locator dann genutzt werden, um einen Teil der Sequenz auszuwählen und ins Anwendungsprogramm (bzw. in die Navigationsbasiskomponente) zu übertragen.

2. Die Typinformationen sind ein integraler Bestandteil einer XQuery-Sequenz. Es sollte deshalb (lesend) auf diese Typinformationen zugegriffen werden können. Dies gilt insbesondere auch für die Typinformationen einer im aktuellen Ergebnistupel enthaltenen Sequenz.

3. Gemäß SQL/XML:2006 kann das aktuelle Ergebnistupel Sequenzen enthalten, die aus *mehreren* Sequenzeinträgen bestehen. Es sollte möglich sein, (lesend) auf diese Sequenzen zuzugreifen.

4. Innerhalb des Ausdrucks, der auf der rechten Seite der `SET`-Klausel einer positionierten `UPDATE`-Anweisung angegeben wird, sollte es nicht erforderlich sein, die aktualisierte Sequenz 'from scratch' zu konstruieren. Stattdessen sollte es genügen, auf die alte Sequenz Bezug zu nehmen und lediglich die vorzunehmenden Änderungen zu spezifieren.

   Ein solches Vorgehen besitzt insbesondere dann Vorteile, wenn an einer großen Sequenz lediglich kleine Änderungen vorgenommen werden sollen. Im Ausdruck, der auf der rechten Seite der `SET`-Klausel genutzt wird, braucht dann die aktualisierte (ebenfalls große) Sequenz nicht aufwendig und umständlich aufgebaut werden — es reicht aus, die wenigen Änderungen kompakt anzugeben.

   Bei dem hier vorgeschlagenen Vorgehen werden innerhalb der positionierten `UPDATE`-Anweisung alle an der Sequenz durchzuführenden Änderungen *explizit* spezifiziert. Dadurch wird es dem DBMS prinzipiell ermöglicht, diese Änderungen „mit spitzen Fingern“ in die Datenbank einzubringen: Statt die alte Sequenz komplett mit der aktualisierten Sequenz überschreiben zu müssen, kann das DBMS einfach die notwendigen Änderungen gezielt in die alte Sequenz einarbeiten.

Beim Einsatz eines *erweiterten SQL/XML:2006-DBMS* — also eines DBMS, das die zuvor beschriebenen (dringend notwendigen) grundlegenden SQL/XML:2006-Erweiterungen unterstützt — ergibt sich die im folgenden Abschnitt betrachtete Architektur.

8.1.3 Prototyparchitektur — Variante 2

Bei der in *Abbildung 8.3* gezeigten Prototyparchitektur sind die zur Realisierung der Sequenzcursor-basierten Verarbeitungsfunktionalität eingeführten Schichten auf ein *erweitertes* SQL/XML:2006-DBMS aufgesetzt — also auf ein DBMS, welches die im vorigen Abschnitt erläuterten grundlegenden SQL/XML:2006-Erweiterungen unterstützt. Die Architektur umfasst insgesamt fünf Schichten:

erweitertes SQL/XML:2006-DBMS
: Das erweiterte SQL/XML:2006-DBMS stellt Zugriffsmöglichkeiten bereit, die es erlauben, adäquat auf die im aktuellen Ergebnistupel enthaltenen Sequenzen zuzugreifen. Nähere Details zu den dabei vorausgesetzten SQL/XML:2006-Erweiterungen finden sich im Abschnitt 8.1.2.

Navigationsbasiskomponente
: Diese Komponente gestattet es der Sequenzcursorverwaltung, lesend auf die typed-value-orientierte Repräsentation der im aktuellen Ergebnistupel enthaltenen Sequenzen zuzugreifen. Dies bedeutet übrigens nicht zwangsläufig, dass die Navigationsbasiskomponente die entsprechenden Sequenzen komplett in die typed-value-orientierte

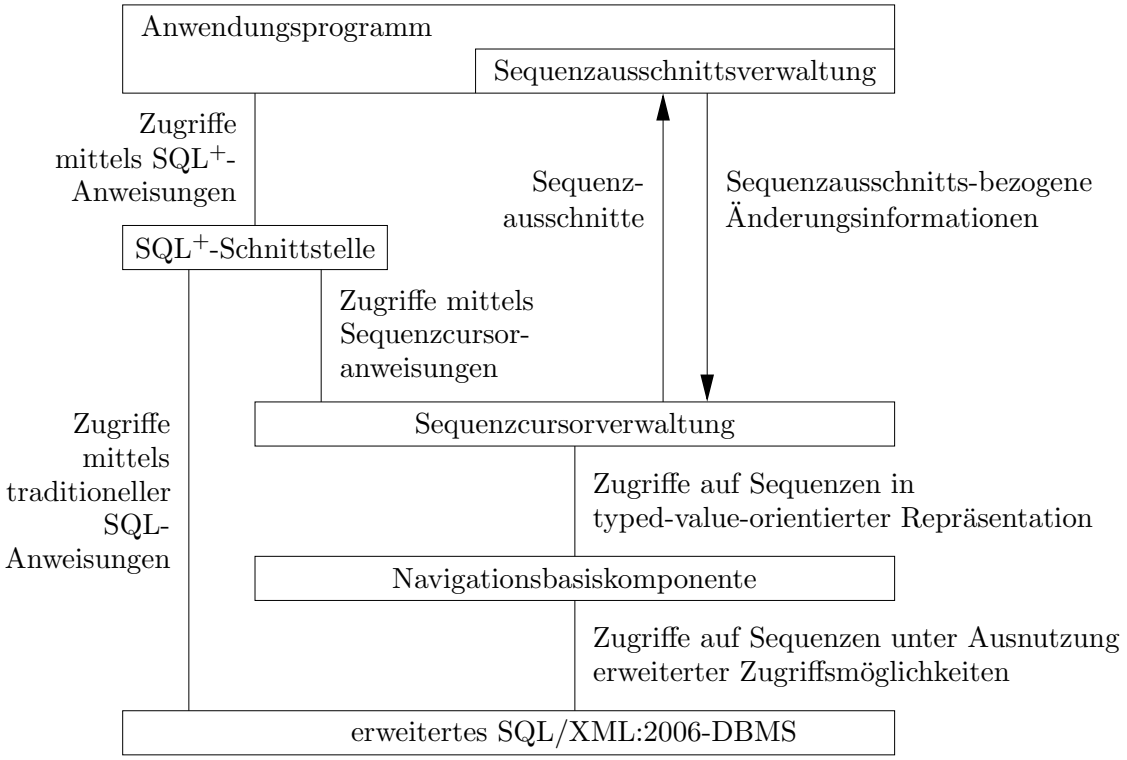


Abbildung 8.3: Prototyparchitektur — Variante 2

Repräsentation überführt — es genügt, die vom Lesezugriff betroffenen Sequenz*teile* zu transformieren [Jat07].

Um eine solche Transformation durchführen zu können, muss die Navigationsbasiskomponente lesend auf die relevanten Teile der im aktuellen Ergebnistupel enthaltenen Sequenzen zugreifen. Die Navigationsbasiskomponente nutzt hierzu die vom erweiterten SQL/XML:2006-DBMS angebotenen (erweiterten) Zugriffsmöglichkeiten.

Eine weitere Aufgabe der Navigationsbasiskomponente besteht darin, Sequenzbezogene Änderungsinformationen von der Sequenzcursorverwaltung entgegenzunehmen und zu verarbeiten. Dadurch wird es ermöglicht, Änderungen in eine im aktuellen Ergebnistupel enthaltene Sequenz (bzw. in die dieser Sequenz zugrunde liegende Ursprungssequenz) einzubringen. Die Navigationsbasiskomponente überführt die entgegengenommenen Änderungsinformationen in eine geeignete positionierte `UPDATE`-Anweisung, welche dann an das erweiterte SQL/XML:2006-DBMS übergeben wird.

Die von der Navigationsbasiskomponente erzeugte positionierte `UPDATE`-Anweisung enthält auf der rechten Seite der `SET`-Klausel einen Ausdruck, in dem alle an der Sequenz vorzunehmenden Änderungen explizit spezifiziert sind [Kie06]. Das erweiterte SQL/XML:2006-DBMS wird somit in die Lage versetzt, die Änderungen „mit spitzen Fingern“ in die Datenbank einzubringen.

Sequenzcursorverwaltung,
SQL$^+$-Schnittstelle und
Anwendungsprogramm mit Sequenzausschnittsverwaltung
Für eine Beschreibung dieser drei Schichten verweisen wir auf Abschnitt 8.1.2.

Eine Umsetzung der hier betrachteten Architektur ist (derzeit) nicht möglich, da (noch) kein DBMS-Produkt verfügbar ist, welches SQL/XML:2006 — bzw. die grundlegenden SQL/XML:2006-Erweiterungen — hinreichend unterstützt: Keins der existierenden DBMS-Produkte ist geeignet, um als unterste Schicht der beschriebenen Architektur zu fungieren. Ausschlaggebend dafür ist (u. a.) das Defizit der heutigen DBMS-Produkte, keine Sequenzen handhaben zu können, die aus *mehreren* Sequenzeinträgen bestehen (Abschnitt 2.12).

Die „Bereitstellung“ eines erweiterten SQL/XML:2006-DBMS ist im Rahmen der vorliegenden Arbeit nicht realisierbar: Aus Aufwandsgründen ist es uns nicht möglich, ein erweitertes SQL/XML:2006-DBMS 'from scratch' zu implementieren. Die Erweiterung eines existierenden DBMS-Produkts um die zusätzlich erforderliche Funktionalität kommt ebenfalls nicht in Frage — der damit verbundene Aufwand wäre zu groß.

In Ermangelung eines erweiterten SQL/XML:2006-DBMS nutzen wir bei der prototypischen Realisierung eine so genannte *Sequenzsimulationskomponente* als unterste Schicht. Diese Sequenzsimulationskomponente [Kie06] simuliert die Möglichkeiten, die ein erweitertes SQL/XML:2006-DBMS für Zugriffe auf die im aktuellen Ergebnistupel enthaltenen Sequenzen bereitstellt. Wir werden im folgenden Abschnitt näher auf die sich damit ergebende (tatsächlich umgesetzte) Prototyparchitektur eingehen.

8.1.4 Prototyparchitektur — Variante 3 (Architektur des implementierten Prototyps)

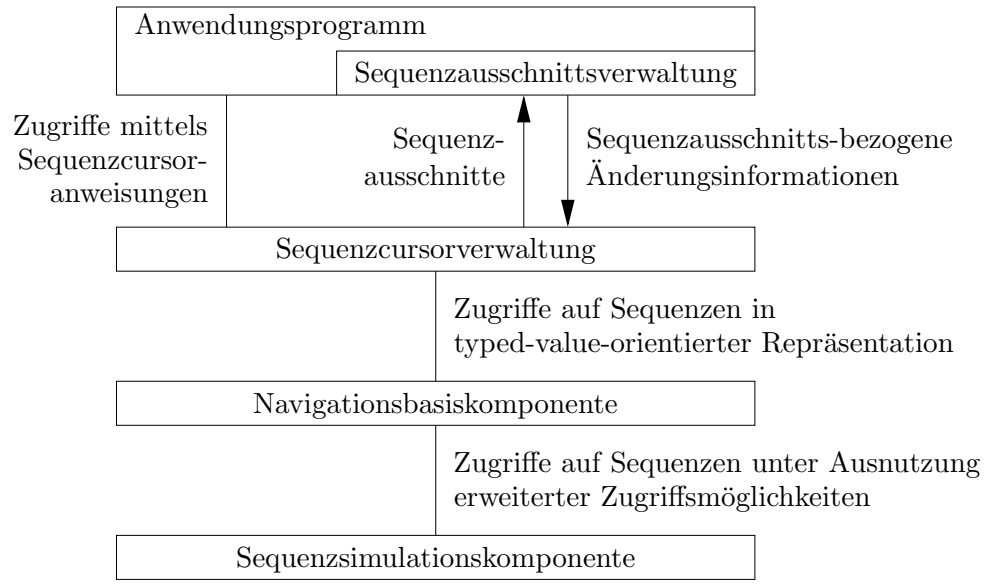


Abbildung 8.4: Architektur des implementierten Prototyps

Abbildung 8.4 zeigt die Architektur des tatsächlich implementierten Prototyps. Es handelt sich dabei um eine Schichtenarchitektur, deren unterste Schicht eine (als „Ersatz“ für das erweiterte SQL/XML:2006-DBMS dienende) Sequenzsimulationskomponente ist. Die Prototyparchitektur besteht aus insgesamt vier Schichten:

Sequenzsimulationskomponente
: Diese Komponente simuliert sowohl die im aktuellen Ergebnistupel enthaltenen Sequenzen als auch die Möglichkeiten, die ein erweitertes SQL/XML:2006-DBMS für (lesende und ändernde) Zugriffe auf diese Sequenzen anbieten würde. Die Sequenzsimulationskomponente ermöglicht es der Navigationsbasiskomponente, lesend auf die Sequenzen zuzugreifen. Außerdem verarbeitet die Sequenzsimulationskomponente positionierte `UPDATE`-Anweisungen, die sie von der Navigationsbasiskomponente entgegennimmt.

  Bei der Sequenzsimulationskomponente handelt es sich *nicht* um ein Datenbankmanagementsystem: Die Sequenzsimulationskomponente kann — abgesehen von positionierten `UPDATE`-Anweisungen — keinerlei SQL-Anweisungen verarbeiten. Folglich können keine SQL-`SELECT`-Anfragen ausgeführt werden und es existiert auch kein „echtes“ Anfrageergebnis und somit auch kein „echtes“ aktuelles Ergebnistupel. Die Simulation der Sequenzen bezieht sich stattdessen auf das *fiktive* aktuelle Ergebnistupel eines *fiktiven* SQL/XML:2006-Anfrageergebnisses.

  Die Beschränkung auf insgesamt *ein* (fiktives) Ergebnistupel ist — für einen Prototyp — durchaus akzeptabel: Viele Arbeitsschritte der Sequenzcursor-basierten Verarbeitung beziehen sich zu jedem Zeitpunkt ohnehin nur auf jeweils *ein* Ergebnistupel — nämlich auf das aktuelle Ergebnistupel. Zu diesen Arbeitsschritten gehört das Positionieren der Sequenzcursor, das Definieren der Sequenzausschnitte, das Übertragen der Sequenzausschnitte, das lokale Arbeiten auf den Sequenzausschnitten sowie das Einbringen bzw. Verwerfen der lokalen Änderungen. Trotz seiner Beschränkung auf ein einziges (fiktives) Ergebnistupel ermöglicht es der Prototyp also, die wesentlichen Konzepte des Sequenzcursor-basierten Verarbeitungsmodells praktisch zu testen.

Navigationsbasiskomponente
: Gegenüber der gleichnamigen Komponente aus Abschnitt 8.1.3 bestehen lediglich folgende Unterschiede:

  - Beim Auslesen der für die Transformation relevanten Sequenzteile greift die Navigationsbasiskomponente auf die Sequenzsimulationskomponente (und nicht auf das erweiterte SQL/XML:2006-DBMS) zu.
  - Die Navigationsbasiskomponente übergibt die von ihr erzeugten positionierten `UPDATE`-Anweisungen an die Sequenzsimulationskomponente (und nicht ans erweiterte SQL/XML:2006-DBMS).

Sequenzcursorverwaltung
: Diese Komponente unterscheidet sich von der gleichnamigen Komponente aus Abschnitt 8.1.2 bzw. Abschnitt 8.1.3 lediglich wie folgt:

  - Die Sequenzcursorverwaltung nimmt die Sequenzcursoranweisungen direkt vom Anwendungsprogramm (und nicht von einer SQL$^+$-Schnittstelle) entgegen.

- Da das Anwendungsprogramm unseres Prototyps zwar Sequenzcursoranweisungen, nicht aber traditionelle SQL-Anweisungen nutzen darf, können weder `FETCH`-Anweisungen noch andere für die Sequenzcursor-basierte Verarbeitung unmittelbar relevante traditionelle SQL-Anweisungen auftreten. Die Sequenzcursorverwaltung braucht folglich nicht auf entsprechende traditionelle SQL-Anweisungen zu reagieren.

Anwendungsprogramm mit Sequenzausschnittsverwaltung
Gegenüber der gleichnamigen Schicht aus Abschnitt 8.1.2 bzw. Abschnitt 8.1.3 ergeben sich ausschließlich die folgenden Unterschiede:

- Das Anwendungsprogramm darf anstelle beliebiger SQL$^+$-Anweisungen lediglich Sequenzcursoranweisungen verwenden.
- Das Anwendungsprogramm übergibt die Sequenzcursoranweisungen direkt an die Sequenzcursorverwaltung (und nicht an eine SQL$^+$-Schnittstelle).

8.2 Implementierung des Prototyps

Der im Rahmen der vorliegenden Arbeit implementierte Prototyp (bei dessen Entwicklung die *nicht*-erweiterte Variante des Sequenzcursor-basierten Verarbeitungsablaufs zugrunde gelegt wurde) besitzt die soeben im Abschnitt 8.1.4 beschriebene — aus vier Schichten bestehende — Architektur. In den sich hier anschließenden Abschnitten 8.2.1 bis 8.2.4 werden wir näher auf die Realisierung der einzelnen Schichten eingehen. Dabei konzentrieren wir uns auf die wichtigsten Aspekte der Implementierung. Für eine umfassende Beschreibung der prototypischen Umsetzung verweisen wir auf [Kie06, Jat07, Böh07, Ott07].

Für die Implementierung des Prototyps wurde die Programmiersprache C++ [Str00, Bre07, Wil07] gewählt, da es sich bei ihr um eine sehr leistungsfähige, objektorientierte Programmiersprache handelt, die über ein gutes Klassenkonzept verfügt. Die von C++ bereitgestellten Sprachmittel ermöglichen eine adäquate Umsetzung der Schichten-weise aufgebauten Prototyparchitektur. Für die Nutzung von C++ spricht außerdem die Verfügbarkeit einer großen Anzahl von Werkzeugen, Bibliotheken und Hilfsprogrammen. Prinzipiell wäre es natürlich auch möglich gewesen, anstelle von C++ eine andere höhere Programmiersprache zu verwenden.

8.2.1 Sequenzsimulationskomponente

Wie aus den Ausführungen des Abschnitts 8.1.4 hervorgeht, besitzt die Sequenzsimulationskomponente im Wesentlichen die folgenden Aufgaben:

- Simulation der Sequenzen, die im fiktiven aktuellen Ergebnistupel eines fiktiven SQL/XML:2006-Anfrageergebnisses enthalten sind
- Bereitstellung von Möglichkeiten zum lesenden Zugriff auf die simulierten Sequenzen
- Verarbeitung von positionierten `UPDATE`-Anweisungen

Bei der Implementierung der Sequenzsimulationskomponente [Kie06] wurden bezüglich des (der Simulation zugrunde liegenden) fiktiven SQL/XML:2006-Anfrageergebnisses folgende vereinfachende Annahmen getroffen:

- Das Anfrageergebnis basiert auf einer (fiktiven) Basistabelle namens T, die u. a. die drei XML-Spalten A, B und C besitzt.
- Dem Anfrageergebnis ist der (fiktive) Tupelcursor $unserTC$ zugeordnet.
- Das Anfrageergebnis umfasst die drei XML-Spalten A, B und C sowie möglicherweise eine oder mehrere Nicht-XML-Spalten. Jeder der drei XML-Ergebnisspalten liegt jeweils die gleichnamige XML-Spalte der Basistabelle T zugrunde.

Die Anzahl und die Namen der (fiktiven) XML-Ergebnisspalten sind also fest in der Sequenzsimulationskomponente „verdrahtet“. Gleiches gilt für den Namen des Tupelcursors und den Namen der Basistabelle. Bei Bedarf ließe sich die Sequenzsimulationskomponente jedoch mit vertretbarem Aufwand so erweitern, dass Flexibilität bezüglich der Anzahl und der Namen der XML-Ergebnisspalten gewährleistet wäre und sich der Tupelcursorname sowie der Basistabellenname frei wählen ließen. Für die meisten mit Hilfe des Prototyps durchzuführenden Untersuchungen ist eine derartige Erweiterung der Sequenzsimulationskomponente allerdings nicht erforderlich.

Simulation der Sequenzen

Die Sequenzsimulationskomponente simuliert die drei im fiktiven aktuellen Ergebnistupel enthaltenen Sequenzen. Jede dieser drei Sequenzen besteht aus mehreren Knoten sowie aus diversen (in der Wurzelebene der Sequenz enthaltenen) atomaren Werten.

Jeder Knoten wird als separates Objekt repräsentiert. Ein solches Knotenobjekt kapselt alle auf den entsprechenden Knoten bezogenen Informationen. Bei diesen (innerhalb der Knotenobjekte gespeicherten) Informationen handelt es sich im Wesentlichen um die vom W3C festgelegten Knoteneigenschaften (Abschnitt 2.10.2.3). Hierzu zählen (in Abhängigkeit von der Knotenart) beispielsweise der Knotentyp, der Knotenname, der textuelle Wert und der getypte Wert. Verweise auf andere Knoten (z. B. Verweise auf den Vaterknoten) werden mit Hilfe von Zeigern realisiert.

Die auf der Wurzelebene der Sequenz vorkommenden atomaren Werte werden ebenfalls mit Hilfe von Objekten repräsentiert. Jedes dieser Objekte enthält sowohl den eigentlichen Wert als auch den Typ(namen) des von ihm repräsentierten atomaren Werts (vgl. Abschnitt 2.10.1).

Bereitstellung von Möglichkeiten zum lesenden Zugriff

Die Sequenzsimulationskomponente stellt verschiedene Methoden bereit, mit deren Hilfe ein gezielter lesender Zugriff auf die Knoten bzw. atomaren Werte der simulierten Sequenzen möglich ist [Kie06]. Für lesende Zugriffe auf die in den Knoten enthaltenen Informationen stehen Methoden zur Verfügung, die an die vom W3C definierten Zugriffsfunktionen (Abschnitt 2.10.2.3) angelehnt sind. So kann beispielsweise mit Hilfe der Methode *node-name* der Name eines Knotens ausgelesen werden, während die Methode *typed-value* den

getypten Wert des Knotens zurückliefert. Soll lesend auf die in einem atomaren Wert enthaltenen Informationen (also auf den eigentlichen Wert bzw. den Typnamen) zugegriffen werden, so sind hierfür die Methoden *value* und *type-name* nutzbar.

Verarbeitung von positionierten `UPDATE`-Anweisungen

Die Sequenzsimulationskomponente kann positionierte `UPDATE`-Anweisungen verarbeiten, bei denen auf der rechten Seite der `SET`-Klausel die in [Kie06] vorgeschlagene SQL-Funktion `XMLMANIPULATE`[3] genutzt wird. Die Verwendung dieser Funktion ist eine Möglichkeit, die letzte der vier im Abschnitt 8.1.2 erläuterten grundlegenden SQL/XML:2006-Erweiterungen zu realisieren.

Mit Hilfe des ersten Parameters der Funktion `XMLMANIPULATE` wird die Sequenz ausgewählt, auf die sich die Änderungen beziehen sollen. Jede an der Sequenz vorzunehmende Änderung wird dann mit jeweils drei aufeinanderfolgenden Parametern spezifiziert:

- Der erste der drei Parameter gibt die Art der vorzunehmenden Änderung an. Beispielsweise drückt `REPLACE CONTENT` aus, dass der Inhalt eines (C- oder T-)Knotens ersetzt werden soll, während sich mit `REPLACE NAME` der Name eines (E- oder A-)Knotens ändern lässt. Ein Ersetzen, Einfügen und Löschen von Knoten ist ebenfalls möglich. Für diesbezügliche Syntaxdetails verweisen wir auf [Kie06].

- Der zweite Parameter enthält die Positionsnummer des Knotens (bzw. des atomaren Werts), auf den sich die Änderungen beziehen. Dabei wird die im Abschnitt 7.4.1.1 vorgestellte Nummerierungsvariante genutzt. Atomare Werte (die auf der Wurzelebene der Sequenz auftreten) werden hierbei behandelt, als würde es sich bei ihnen um Knoten handeln. Ist beispielsweise der fünfte Sequenzeintrag ein atomarer Wert, so besitzt dieser die Positionsnummer *5*. Handelt es sich beim fünften Sequenzeintrag hingegen um einen Baum, so identifiziert die Positionsnummer *5* dessen Wurzelknoten. Der siebente Kindknoten dieses Wurzelknotens hat dann die Positionsnummer *5.7*. Die Positionsnummer *5.7.–76* gehört wiederum zum 76. diesem Kindknoten zugeordneten A-Knoten usw.

- Die Bedeutung des dritten Parameters hängt von der Art der durchzuführenden Änderung ab. Soll beispielsweise der Inhalt eines Knotens ersetzt werden, spezifiziert dieser Parameter den neuen Knoteninhalt. Soll hingegen der Knotenname geändert werden, wird mit Hilfe des Parameters der neue Knotenname angegeben.

Bei einem einzelnen Aufruf der Funktion `XMLMANIPULATE` können gleichzeitig *mehrere* Änderungen spezifiziert werden — die verschiedenen Änderungen werden hierzu einfach hintereinander (jeweils mit Hilfe von drei Parametern) angegeben [Kie06].

Abbildung 8.5 zeigt ein Beispiel für eine positionierte `UPDATE`-Anweisung, bei der die SQL-Funktion `XMLMANIPULATE` zum Einsatz kommt. Mit Hilfe dieser `UPDATE`-Anweisung werden zwei Änderungen an der (in der Ergebnisspalte *B* enthaltenen) simulierten Sequenz vorgenommen: Zum einen wird der Name des durch die Positionsnummer *21.5.–77* identifizierten A-Knotens in *Geburtstag* abgeändert, zum anderen wird der bisherige Inhalt des Knotens *3.2.6* durch *Jena* ersetzt.

[3] Diese Funktion ist an die in Oracle 11g verfügbare SQL-Funktion `UPDATEXML` [ORA07c] angelehnt.

```
UPDATE T
SET B = XMLMANIPULATE(B,
                      REPLACE NAME, 21.5.-77, 'Geburtstag',
                      REPLACE CONTENT, 3.2.6, 'Jena')
WHERE CURRENT OF unserTC;
```

Abbildung 8.5: Beispiel für eine positionierte UPDATE-Anweisung

Der Sequenzsimulationskomponente werden die von ihr zu verarbeitenden positionierten UPDATE-Anweisungen in Form von Zeichenketten übergeben. Da bezüglich des Basistabellennamens und des Tupelcursornamens keine Flexibilität möglich ist und als Spaltenname nur A, B oder C auftreten darf, gestaltet sich das (von der Sequenzsimulationskomponente durchzuführende) Parsen der Zeichenketten relativ einfach: Die Arbeit beschränkt sich im Wesentlichen darauf, die innerhalb der runden Klammern auftretenden Parameterwerte auszuwerten.

Die (eigentliche) Ausführung einer positionierten UPDATE-Anweisung besteht darin, dass die Sequenzsimulationskomponente die spezifizierten Änderungen nacheinander (also Änderung für Änderung) in diejenige der drei simulierten Sequenzen einarbeitet, auf die innerhalb der positionierten UPDATE-Anweisung Bezug genommen wird. Nachdem alle Änderungen eingebracht wurden, gibt die Sequenzsimulationskomponente das Ergebnis des Änderungseinbringens (also die geänderte simulierte Sequenz) auf dem Bildschirm aus.[4] Dabei werden für jeden in der Sequenz enthaltenen Knoten bzw. atomaren Wert die wichtigsten Eigenschaften aufgelistet. Für atomare Werte sind dies die Positionsnummer, der eigentliche Wert sowie der Typname. Bei Knoten erfolgt eine Ausgabe der Positionsnummer und der Knotenart. Abhängig von der Knotenart können zudem weitere Eigenschaften ausgegeben werden — beispielsweise der Knoteninhalt bei C- oder T-Knoten bzw. der Knotentyp und der Knotenname bei E- oder A-Knoten.

8.2.2 Navigationsbasiskomponente

Wie im Abschnitt 8.1.4 bzw. 8.1.3 geschildert, ist die Navigationsbasiskomponente im Wesentlichen für folgende Aufgaben verantwortlich:

- Bereitstellung von Möglichkeiten zum lesenden Zugriff auf die typed-value-orientierte Repräsentation der simulierten Sequenzen
- Überführung von Sequenz-bezogenen Änderungsinformationen in positionierte UPDATE-Anweisungen

Bereitstellung von Möglichkeiten zum lesenden Zugriff

Die Navigationsbasiskomponente [Jat07] stellt Methoden bereit, mit deren Hilfe (lesend) auf die simulierten Sequenzen zugegriffen werden kann, als würden diese Sequenzen in typed-value-orientierter Repräsentation vorliegen. Hierzu überführt die Navigationsbasiskomponente die simulierten Sequenzen (bzw. Teile dieser Sequenzen) in die typed-value-orientierte Repräsentation.

[4] Die Ausgabe kann bei Bedarf in eine Datei umgeleitet werden.

Während eine traditionell repräsentierte Sequenz atomare Werte als Sequenzeinträge enthalten kann, setzt sich eine in typed-value-orientierter Repräsentation vorliegende Sequenz ausschließlich aus Bäumen zusammen, die wiederum aus Knoten bestehen (Abschnitt 3.8.1). Diese Knoten werden innerhalb der Navigationsbasiskomponente mit Hilfe von Objekten repräsentiert. Diese Objekte kapseln alle Knoten-bezogenen Informationen, wie z. B. die Knotenart, den Knotentyp oder den Knotennamen. Beziehungen zwischen Knoten werden mit Hilfe von Zeigern realisiert. So besitzt ein Knotenobjekt beispielsweise einen Zeiger auf den Vater, einen Zeiger auf den linkesten Sohn sowie Zeiger auf den linken bzw. rechten Bruder.

Um einen effektiven Zugriff auf die einzelnen in der Sequenz enthaltenen Bäume zu gewährleisten, existiert ein Zeiger-Array, welches Zeiger auf die Baumwurzeln enthält. Der in der i-ten Komponente dieses Arrays enthaltene Zeiger verweist dabei auf das Objekt, das den Wurzelknoten des i-ten Baums der Sequenz repräsentiert. *Abbildung 8.6* veranschaulicht die Objekt- und Zeiger-basierte Repräsentation einer (in typed-value-orientierter Repräsentation vorliegenden) Beispielsequenz.

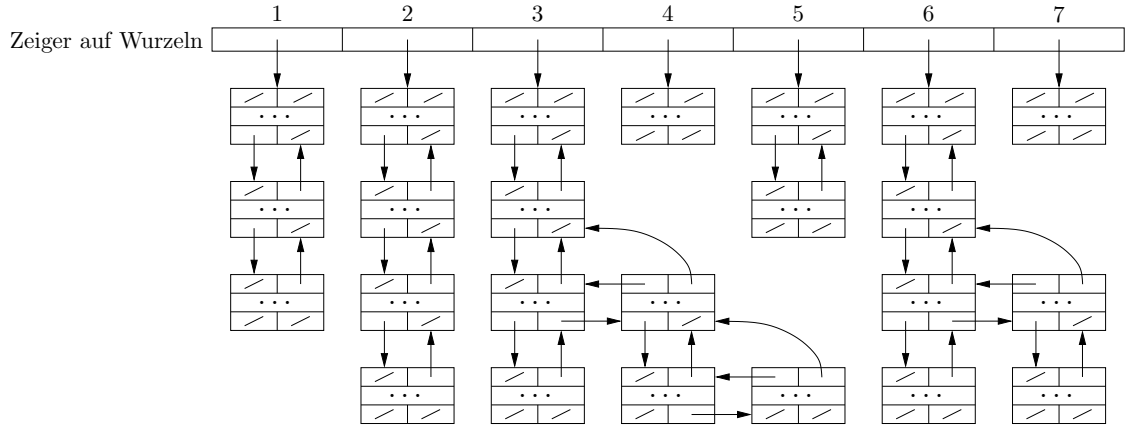


Abbildung 8.6: Objekt- und Zeiger-basierte Repräsentation (vereinfacht)

Die von der Navigationsbasiskomponente bereitgestellten Methoden erlauben es, gezielt auf die Eigenschaften der einzelnen Knoten zuzugreifen. Die Bezugnahme auf einen bestimmten Knoten erfolgt dabei über die Kombination aus Ergebnisspaltenname (A, B oder C) und Knotennummer (z. B. *3.2.6*). Die Knotennummer bezieht sich hierbei auf die typed-value-orientierte Repräsentation der Sequenz (und *nicht* auf die traditionelle Sequenzrepräsentation). Es wird die aus Abschnitt 7.4.1.1 bekannte Knotennummerierungsvariante zugrunde gelegt.

Mit Hilfe des Methodenaufrufs *node-name(“A“, “3.2.6“)* kann beispielsweise der Name des Knotens mit der Knotennummer *3.2.6* ermittelt werden. Die Namensermittlung bezieht sich dabei auf die typed-value-orientierte Repräsentation der in der Ergebnisspalte A enthaltenen simulierten Sequenz. Eine Abfrage der Knotenart wäre mittels des Methodenaufrufs *node-kind(“A“, “3.2.6“)* möglich. Zum Auslesen der anderen Knoteneigenschaften stehen ebenfalls geeignete Zugriffsmethoden zur Verfügung.

Die Navigationsbasiskomponente überführt i. d. R. nur *Teile* der simulierten Sequenzen (also nicht die kompletten Sequenzen) in die typed-value-orientierte Repräsentation. Die Transformation eines Sequenzteils erfolgt dabei erst dann, wenn (mit Hilfe der von der

Navigationsbasiskomponente bereitgestellten Zugriffsmethoden) erstmals auf die typed-value-orientierte Repräsentation des entsprechenden Sequenzteils zugegriffen werden soll:

- Ein sich auf der Wurzelebene einer simulierten Sequenz befindender atomarer Wert wird in die typed-value-orientierte Repräsentation überführt, sobald (erstmals) auf seine typed-value-orientierte Repräsentation zugegriffen werden soll. Die Überführung des atomaren Werts erfolgt wie im Abschnitt 3.6 beschrieben.
- Ein Knoten, der sich auf der Wurzelebene einer simulierten Sequenz befindet, wird in die typed-value-orientierte Repräsentation überführt, sobald (erstmals) ein Zugriff auf seine typed-value-orientierte Repräsentation erfolgen soll. Für Details bezüglich der Überführung verweisen wir auf die Abschnitte 3.3 bis 3.5.
- Ein aus mehreren Knoten bestehender Sequenzeintrag einer simulierten Sequenz wird komplett in die typed-value-orientierte Repräsentation überführt, sobald (erstmals) auf einen Nicht-Wurzelknoten seiner typed-value-orientierten Repräsentation zugegriffen werden soll. Einzelheiten zur Durchführung der Überführung finden sich in den Abschnitten 3.3 bis 3.5.

Beim (schrittweisen) Aufbau der typed-value-orientierten Repräsentation greift die Navigationsbasiskomponente lesend auf die Knoten und atomaren Werte der simulierten Sequenzen zu. Hierzu nutzt die Navigationsbasiskomponente die ihr von der Sequenzsimulationskomponente bereitgestellten Methoden.

Für jede der drei simulierten Sequenzen wird mit Hilfe von zwei Boolean-Arrays der Fortschritt der Überführung protokolliert: Die i-te Komponente des einen Arrays enthält genau dann den Wert *true*, wenn der i-te Sequenzeintrag *komplett* in die typed-value-orientierte Repräsentation überführt wurde. In der i-ten Komponente des anderen Arrays ist verzeichnet, ob zumindest eine *teilweise* Transformation des i-ten Sequenzeintrags stattgefunden hat. Wird nun (mit Hilfe einer von der Navigationsbasiskomponente bereitgestellten Zugriffsmethode) versucht, lesend auf die Eigenschaften eines bestimmten Knotens zuzugreifen, so kann die Navigationsbasiskomponente unter Zuhilfenahme der beiden Arrays sehr leicht ermitteln, ob der entsprechende Knoten bereits in der (bisher aufgebauten) typed-value-orientierten Repräsentation enthalten ist oder ob infolge des Lesezugriffs die Transformation eines Sequenzteils notwendig wird.

Wir werden die schrittweise Erzeugung der typed-value-orientierten Repräsentation kurz anhand eines Beispiels veranschaulichen. Wir beziehen uns dabei auf diejenige simulierte Sequenz, die in der Ergebnisspalte A enthalten ist.

Sofern bisher noch kein Zugriff auf die typed-value-orientierte Repräsentation der Sequenz erfolgte, hat die Navigationsbasiskomponente auch noch nicht mit der Transformation der Sequenz begonnen *(Abbildung 8.7)*.

	1	2	3	4	5	6	7
komplett?	false	false	false	false	false	false	false
zumindest teilweise?	false	false	false	false	false	false	false
Zeiger auf Wurzeln	⁄	⁄	⁄	⁄	⁄	⁄	⁄

Abbildung 8.7: Repräsentation vor dem ersten Zugriff

Wird nun mittels der Methodenaufrufe *node-kind(“A“, “2“)*, *node-kind(“A“, “3“)* und *node-kind(“A“, “6“)* auf die Wurzelebene der typed-value-orientierten Repräsentation zugegriffen, werden die den entsprechenden Knoten zugrunde liegenden (sich ebenfalls auf der Wurzelebene befindenden) Knoten bzw. atomaren Werte in die typed-value-orientierte Repräsentation überführt *(Abbildung 8.8)*.

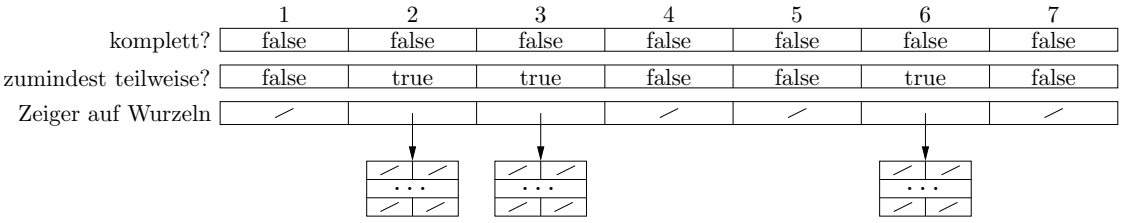


Abbildung 8.8: Repräsentation nach drei Zugriffen auf die Wurzelebene

Erfolgt nun mit Hilfe des Methodenaufrufs *node-kind(“A“, “3.1.1“)* ein Zugriff auf einen Nicht-Wurzelknoten des dritten Baums der typed-value-orientierten Repräsentation, so wird der komplette diesem Baum zugrunde liegende Sequenzeintrag in die typed-value-orientierte Repräsentation überführt *(Abbildung 8.9)*.

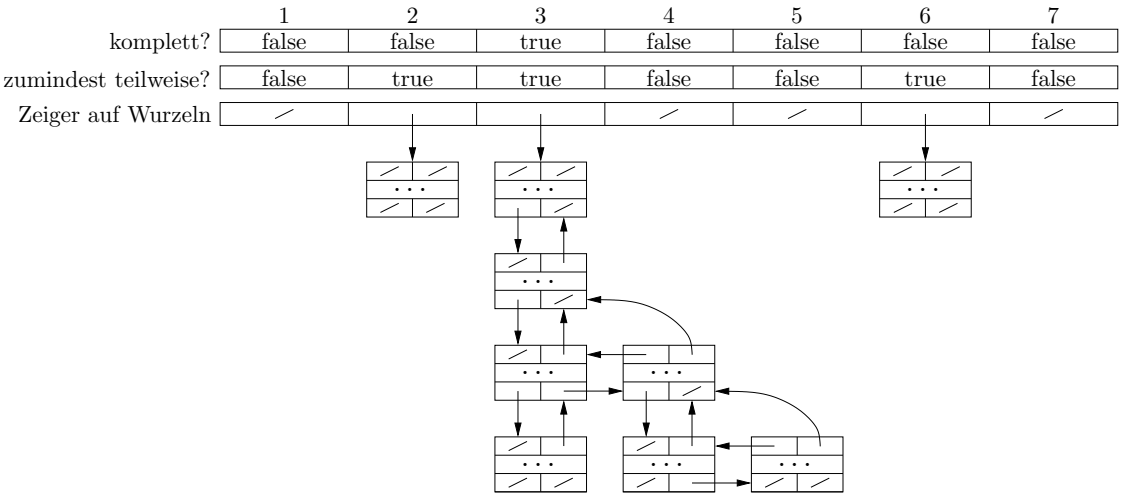


Abbildung 8.9: Repräsentation nach zusätzlichem Zugriff auf inneren Knoten

Überführung von Sequenz-bezogenen Änderungsinformationen

Außer für das Bereitstellen von Möglichkeiten zum lesenden Zugriff auf die typed-value-orientierte Repräsentation der simulierten Sequenzen ist die Navigationsbasiskomponente auch dafür verantwortlich, Sequenz-bezogene Änderungsinformationen in positionierte `UPDATE`-Anweisungen zu überführen. Diese positionierten `UPDATE`-Anweisungen (welche auf der rechten Seite der `SET`-Klausel einen Aufruf der Funktion `XMLMANIPULATE` enthalten) werden dann in Form von Zeichenketten an die Sequenzsimulationskomponente übergeben.

Bei den auf eine bestimmte Sequenz bezogenen Änderungsinformationen (welche die Navigationsbasiskomponente von der Sequenzcursorverwaltung entgegennimmt) handelt es sich

um eine Folge einzelner Änderungseinträge, wobei jeder dieser Änderungseinträge jeweils mit einem geänderten, gelöschten oder neu eingefügten Knoten korrespondiert [Jat07]. Ein solcher Änderungseintrag bezieht sich auf die typed-value-orientierte Repräsentation der Sequenz und enthält sämtliche Informationen, die für das Ändern, Löschen bzw. Einfügen des entsprechenden Knotens notwendig sind. Zu diesen Informationen gehört beispielsweise die Positionsnummer des betroffenen Knotens sowie die genaue Art der vorzunehmenden Änderung.

Die Änderungseinträge werden nacheinander (also Änderungseintrag für Änderungseintrag) von der Navigationsbasiskomponente verarbeitet. Die Navigationsbasiskomponente ermittelt dabei jeweils für jeden Änderungseintrag, wie sich die entsprechende Änderung auf die *traditionelle* Repräsentation der Sequenz auswirkt [Jat07]. Die Navigationsbasiskomponente überführt anschließend jede an der traditionell repräsentierten Sequenz vorzunehmende Änderung in jeweils drei Parameter, die dann innerhalb der von der Navigationsbasiskomponente erzeugten positionierten `UPDATE`-Anweisung (und zwar beim Aufruf der `XMLMANIPULATE`-Funktion) dazu genutzt werden, die entsprechende Änderung zu beschreiben. Für nähere Details verweisen wir auf [Jat07].

8.2.3 Sequenzcursorverwaltung

Wie aus Abschnitt 8.1.4 (bzw. aus Abschnitt 8.1.2) hervorgeht, hat die Sequenzcursorverwaltung im Wesentlichen folgende Aufgaben:

- Verwaltung der Sequenzcursor
- Verwaltung der Sequenzausschnittsbeschreibungsdaten
- Verarbeitung von Sequenzcursoranweisungen

Verwaltung der Sequenzcursor

Die Sequenzcursorverwaltung [Böh07, Ott07] hält für jeden Sequenzcursor die folgenden Informationen vor:

- Name des Sequenzcursors
- Name der Ergebnisspalte, an die der Sequenzcursor gebunden ist (*„A“*, *„B“* oder *„C“*)
- Status des Sequenzcursors (*'P'* oder *'N'*)[5]

Für jeden Sequenzcursor, der sich im Status *„positioniert“* befindet, wird zudem die Positionsnummer desjenigen Knotens vermerkt, auf dem der entsprechende Sequenzcursor aktuell steht. Dabei wird (wieder) die aus Abschnitt 7.4.1.1 bekannte Knotennummerierungsvariante zugrunde gelegt.

[5] Wie bereits erwähnt, basiert die Implementierung des Prototyps auf der *nicht*-erweiterten Variante des Sequenzcursor-basierten Verarbeitungsablaufs. Somit existieren für den Status eines Sequenzcursors nur die Möglichkeiten *„positioniert“ ('P')* und *„unpositioniert“ ('N')*.

Die auf einen Sequenzcursor bezogenen Informationen werden erzeugt, sobald der entsprechende Sequenzcursor mit Hilfe der Sequenzcursoranweisung `CREATE SEQUENCE CURSOR` definiert wird. Eine Ausführung der Sequenzcursoranweisung `MOVE SEQUENCE CURSOR` bewirkt eine Aktualisierung der Status- bzw. Positionsinformation des betroffenen Sequenzcursors. Das (mittels der Sequenzcursoranweisung `DROP SEQUENCE CURSOR` durchgeführte) Löschen eines Sequenzcursors hat zur Folge, dass die auf diesen Sequenzcursor bezogenen Informationen ebenfalls gelöscht werden.

Verwaltung der Sequenzausschnittsbeschreibungsdaten

Für jeden (aktuell definierten) Sequenzausschnitt werden von der Sequenzcursorverwaltung folgende Informationen vorgehalten:

- Name der Ergebnisspalte, die den Sequenzausschnitt enthält (*„A“*, *„B“* oder *„C“*)
- Status des Sequenzausschnitts (*1*, *2*, *3* oder *4*)
  - *1* — Der Sequenzausschnitt wurde noch nicht ins Anwendungsprogramm übertragen.
  - *2* — Der Sequenzausschnitt wurde ins Anwendungsprogramm übertragen. Die lokalen Änderungen wurden (bisher) weder eingebracht noch verworfen.
  - *3* — Die lokalen Änderungen wurden eingebracht.
  - *4* — Die lokalen Änderungen wurden verworfen.
- Anzahl der Sequenzausschnittsteile, aus denen sich der Sequenzausschnitt zusammensetzt
- Informationen über die einzelnen Sequenzausschnittsteile, die zum Sequenzausschnitt gehören

Zu den bezüglich der einzelnen Sequenzausschnittsteile gespeicherten Informationen gehören die folgenden Angaben:

- Nummer des Sequenzausschnittsteils gemäß der im Abschnitt 6.1.4 beschriebenen Nummerierung
- SAT-Klasse, welcher der Sequenzausschnittsteil zugeordnet ist (vgl. Abschnitt 6.2)

In Abhängigkeit von der konkreten SAT-Klasse werden über einen Sequenzausschnittsteil weitere Informationen vorgehalten. Für einen Sequenzausschnittsteil der SAT-Klasse *5* handelt es sich dabei beispielsweise um die Positionsnummern der beiden Wurzelknoten, die den Sequenzausschnittsteil nach links bzw. rechts begrenzen.

Die Sequenzausschnitts-bezogenen Informationen werden bei der Ausführung der Sequenzcursoranweisung `DEFINE SEQUENCE PART` erzeugt und infolge der Ausführung der Sequenzcursoranweisung `UNDEFINE SEQUENCE PART` wieder gelöscht. Ein Ausführen der Sequenzcursoranweisung `TRANSFER SEQUENCE PART`, `BRING IN LOCAL CHANGES` bzw. `DISCARD LOCAL CHANGES` bewirkt eine Aktualisierung der Statusinformation des betroffenen Sequenzausschnitts.

Verarbeitung von Sequenzcursoranweisungen

Der Sequenzcursorverwaltung wird eine von ihr zu verarbeitende Sequenzcursoranweisung in Form einer Zeichenkette übergeben. Diese Zeichenkette wird von der Sequenzcursorverwaltung zunächst geparst. Die weitere Verarbeitung hängt dann davon ab, um welche Sequenzcursoranweisung es sich handelt.

Wir werden im Folgenden kurz auf die Verarbeitung der einzelnen Sequenzcursoranweisungen eingehen. Für nähere Details verweisen wir auf [Böh07, Ott07].

`CREATE SEQUENCE CURSOR[S]`
: Die Sequenzcursorverwaltung erzeugt die bezüglich der entsprechenden Sequenzcursor vorzuhaltenden Informationen. Welche Informationen dies im Einzelnen sind, haben wir am Anfang des aktuellen Abschnitts beschrieben.

`DROP SEQUENCE CURSOR[S]`
: Die Sequenzcursorverwaltung löscht die Informationen, die von ihr bezüglich der entsprechenden Sequenzcursor vorgehalten werden.

`MOVE SEQUENCE CURSOR`
: Die Sequenzcursorverwaltung ermittelt zunächst die Positionsnummer des Knotens, auf den der Sequenzcursor positioniert werden soll. Dies ist oftmals möglich, *ohne* dass dazu ein Zugriff auf die entsprechende Sequenz erforderlich ist. Besitzt ein Sequenzcursor, der (mittels `MOVE meinSC TO PARENT`) zum Vaterknoten bewegt werden soll, *vor* der Positionierung beispielsweise die Positionsinformation *15.1.51*, so lautet seine Positionsinformation *nach* der Positionierung *15.1.* Die von der Sequenzcursorverwaltung bezüglich des Sequenzcursors vorgehaltenen Informationen reichen im konkreten Beispiel also aus, um die neue Positionsinformation zu bestimmen. Es existieren allerdings auch Fälle, in denen zur Ermittlung der neuen Positionsinformation ein Zugriff auf die von der Navigationsbasiskomponente bereitgestellten Informationen notwendig ist. Ein solcher Fall liegt z. B. vor, wenn bei der Positionierung die Knotenart, der Knotentyp oder der Knotenname berücksichtigt werden soll.

  Nachdem die Ermittlung der neuen Positionsinformation abgeschlossen ist, aktualisiert die Sequenzcursorverwaltung die bezüglich des Sequenzcursors vorgehaltene Positionsinformation entsprechend. Außerdem führt sie eine Aktualisierung der (den Sequenzcursor betreffenden) Statusinformation durch.

  Anschließend nutzt die Sequenzcursorverwaltung die (neue) Positionsinformation, um (mit Hilfe der von der Navigationsbasiskomponente zur Verfügung gestellten Methoden) lesend auf die Eigenschaften desjenigen Knotens zuzugreifen, auf dem der Sequenzcursor nun steht. Ein Auslesen dieser Knoteneigenschaften ist erforderlich, um die SQLSCFA aktualisieren zu können.

`GET INFORMATION ABOUT SEQUENCE CURSOR`
: Die Sequenzcursorverwaltung führt eine Aktualisierung der SQLSCFA durch. Hierzu nutzt sie (u. a.) die Informationen, die von ihr bezüglich des entsprechenden Sequenzcursors vorgehalten werden.

  Sofern der entsprechende Sequenzcursor den Status *„positioniert“* besitzt, greift die Sequenzcursorverwaltung außerdem (lesend) auf die Eigenschaften des Knotens zu,

auf dem der Sequenzcursor aktuell steht. Diese Knoteneigenschaften werden für die Aktualisierung der SQLSCFA benötigt.

`DEFINE SEQUENCE PART`
: Die Sequenzcursorverwaltung erzeugt die bezüglich des entsprechenden Sequenzausschnitts vorzuhaltenden Informationen. Um welche Informationen es sich dabei handelt, haben wir im aktuellen Abschnitt bereits beschrieben.

`UNDEFINE SEQUENCE PART`
: Die Sequenzcursorverwaltung löscht die von ihr bezüglich des entsprechenden Sequenzausschnitts vorgehaltenen Informationen.

`DESCRIBE SEQUENCE PART`
: Im Rahmen unserer prototypischen Realisierung erfolgt die lokale Repräsentation eines Sequenzausschnitts mit Hilfe von Objekten, die alle dieselbe Größe besitzen. Jeder im Sequenzausschnitt enthaltene Knoten wird dabei als separates Objekt repräsentiert.

  Unter Zuhilfenahme der über den entsprechenden Sequenzausschnitt vorgehaltenen Informationen sowie mit Hilfe der von der Navigationsbasiskomponente angebotenen Methoden ermittelt die Sequenzcursorverwaltung zunächst, aus wie vielen Knoten sich der betreffende Sequenzausschnitt zusammensetzt. Aus der Anzahl der Knoten berechnet die Sequenzcursorverwaltung dann, wie groß ein lokaler Speicherbereich mindestens sein muss, um den Sequenzausschnitt komplett aufnehmen zu können. Anschließend vermerkt die Sequenzcursorverwaltung die entsprechende Größenangabe in der SQLSPCA.

`TRANSFER SEQUENCE PART`
: Bei unserer prototypischen Umsetzung erfolgt die lokale Repräsentation eines Sequenzausschnitts Objekt- und Zeiger-basiert: Jeder im Sequenzausschnitt enthaltene Knoten wird als separates Objekt repräsentiert, wobei die Verweise zwischen den Objekten mit Hilfe von Zeigern realisiert werden (vgl. Abschnitt 6.4.1).

  Die gewünschte lokale Hauptspeicherrepräsentation des zu übertragenden Sequenzausschnitts wird von der Sequenzcursorverwaltung zunächst komplett „vorberechnet“ (vgl. Abschnitt 6.4.2) und dann — in einem einzigen Schritt — in den (zur Aufnahme des Sequenzausschnitts vorgesehenen) lokalen Speicherbereich kopiert. Die Adresse dieses Speicherbereichs entnimmt die Sequenzcursorverwaltung dabei der SQLSPCA. Für nähere Details verweisen wir auf [Ott07].

  Im Anschluss an die Übertragung des Sequenzausschnitts aktualisiert die Sequenzcursorverwaltung die bezüglich dieses Sequenzausschnitts vorgehaltene Statusinformation.

`BRING IN LOCAL CHANGES`
: Die Sequenzcursorverwaltung greift zunächst lesend auf die SQLSPCA zu, um sowohl die Adresse als auch die Größe des lokalen Speicherbereichs zu ermitteln, in dem die Sequenzausschnitts-bezogenen Änderungsinformationen enthalten sind. Anschließend liest die Sequenzcursorverwaltung die Änderungsinformationen aus diesem Speicherbereich aus.

  Bei den auf einen Sequenzausschnitt bezogenen Änderungsinformationen handelt es sich um eine Folge von Änderungseinträgen, wobei jeder dieser Änderungseinträge

mit einem lokal geänderten, lokal gelöschten oder lokal neu eingefügten Knoten korrespondiert. Die Sequenzcursorverwaltung überführt nun diese Sequenzausschnittsbezogenen Änderungseinträge in Änderungseinträge, die sich auf die Gesamtsequenz beziehen. Hierzu verarbeitet die Sequenzcursorverwaltung einen Sequenzausschnittsbezogenen Änderungseintrag nach dem anderen und ermittelt dabei jeweils, wie sich die entsprechende (lokale) Änderung auf die Gesamtsequenz auswirkt.

Die Sequenzcursorverwaltung übergibt die Gesamtsequenz-bezogenen Änderungseinträge dann an die Navigationsbasiskomponente. Außerdem aktualisiert die Sequenzcursorverwaltung die Statusinformation, die sie bezüglich des entsprechenden Sequenzausschnitts vorhält.

`DISCARD LOCAL CHANGES`
: Die Sequenzcursorverwaltung aktualisiert die Statusinformation des betroffenen Sequenzausschnitts.

8.2.4 Anwendungsprogramm mit Sequenzausschnittsverwaltung

Abschnitt 8.2.4.1 beschäftigt sich zunächst mit dem (eigentlichen) Anwendungsprogramm. Die Sequenzausschnittsverwaltung ist anschließend Gegenstand von Abschnitt 8.2.4.2.

8.2.4.1 Anwendungsprogramm

Das Anwendungsprogramm arbeitet — mit gewissen Einschränkungen — gemäß des Sequenzcursor-basierten Verarbeitungsablaufs. Die auszuführenden Sequenzcursoranweisungen übergibt das Anwendungsprogramm dabei direkt an die Sequenzcursorverwaltung.

Die Übergabe einer Sequenzcursoranweisung an die Sequenzcursorverwaltung erfolgt mit Hilfe der Methode *executeSCS()*[6]. Dieser Methode wird die auszuführende Sequenzcursoranweisung in Form einer Zeichenkette übergeben. Soll beispielsweise der Sequenzcursor *meinSC* auf den Wurzelknoten des letzten Baums positioniert werden, so kann dies durch den in *Abbildung 8.10* gezeigten Methodenaufruf veranlasst werden.

```
executeSCS("MOVE meinSC TO ROOT OF LAST TREE");
```

Abbildung 8.10: Übergabe einer Sequenzcursoranweisung

Wie bereits erwähnt, unterliegt unsere prototypische Umsetzung gewissen Beschränkungen. So darf z. B. das Anwendungsprogramm anstelle beliebiger SQL$^+$-Anweisungen lediglich Sequenzcursoranweisungen verwenden (Abschnitt 8.1.4). Außerdem existiert nur ein einziges (fiktives) Ergebnistupel. Infolge dieser Beschränkungen lassen sich drei der im Abschnitt 4.2 beschriebenen Verarbeitungsschritte *nicht* mit Hilfe unseres Prototyps realisieren. Es handelt sich dabei um folgende Arbeitsschritte:

1. Festlegen der `SELECT`-Anfrage und Anfrageausführung
2. Weiterbewegen des Tupelcursors

[6] Das im Methodenname enthaltene Kürzel *SCS* steht für *sequence cursor statement*, also *Sequenzcursoranweisung*.

3. Schließen des Tupelcursors

Die (im Rahmen unserer prototypischen Umsetzung) vom Anwendungsprogramm durchgeführte Verarbeitung umfasst somit ausschließlich die folgenden sieben Verarbeitungsschritte:

1. Erzeugen der Sequenzcursor (Abschnitt 4.2.2)
2. Positionieren der Sequenzcursor (Abschnitt 4.2.4)
3. Definieren der Sequenzausschnitte (Abschnitt 4.2.5)
4. Übertragen der Sequenzausschnitte (Abschnitt 4.2.6)
5. Lokales Arbeiten auf den Sequenzausschnitten (Abschnitt 4.2.7)
6. Einbringen oder Verwerfen der lokalen Änderungen (Abschnitt 4.2.8)
7. Löschen der Sequenzcursor (Abschnitt 4.2.9)

Für nähere Details bezüglich der einzelnen Verarbeitungsschritte verweisen wir auf die angegebenen Abschnitte.

8.2.4.2 Sequenzausschnittsverwaltung

Wie im Abschnitt 8.1.4 mit Bezug auf Abschnitt 8.1.2 geschildert, besitzt die Sequenzausschnittsverwaltung [Ott07] im Wesentlichen die folgenden Aufgaben:

- Lokales Verwalten der ins Anwendungsprogramm übertragenen Sequenzausschnitte
- Bereitstellung von Möglichkeiten zum (lesenden und ändernden) lokalen Arbeiten auf den Sequenzausschnitten
- Protokollierung der lokal an den Sequenzausschnitten vorgenommenen Änderungen
- Bereitstellung sämtlicher auf einen Sequenzausschnitt bezogenen Änderungsinformationen in einem zusammenhängenden lokalen Speicherbereich

Lokales Verwalten der Sequenzausschnitte

Jeder ins Anwendungsprogramm übertragene Sequenzausschnitt wird lokal mit Hilfe eines Objekts der Klasse *SPM (sequence part manager)* verwaltet (Abschnitt 4.2.6). Diese Klasse stellt dem Anwendungsprogramm u. a. die folgenden vier Methoden zur Verfügung:

get_pointer_to_allocated_storage()
: Mit Hilfe dieser Methode kann die Adresse des lokalen Speicherbereichs abgefragt werden, der zur Aufnahme des Sequenzausschnitts vorgesehen ist.

provide_change_information()
: Mittels dieser Methode wird veranlasst, dass sämtliche den entsprechenden Sequenzausschnitt betreffenden Änderungsinformationen in einem zusammenhängenden lokalen Speicherbereich bereitgestellt werden.

get_pointer_to_change_information()
: Sofern zuvor die Methode *provide_change_information()* aufgerufen wurde, liefert die Methode *get_pointer_to_change_information()* die Adresse des lokalen Speicherbereichs, in dem die Sequenzausschnitts-bezogenen Änderungsinformationen bereitgestellt wurden. Andernfalls wird ein NULL-Zeiger zurückgeliefert.

get_size_of_change_information()
: Einen bereits erfolgten Aufruf der Methode *provide_change_information()* vorausgesetzt, lässt sich mit Hilfe der Methode *get_size_of_change_information()* ermitteln, wie groß der lokale Speicherbereich ist, der die Sequenzausschnitts-bezogenen Änderungsinformationen enthält. Sollten die entsprechenden Änderungsinformationen hingegen noch nicht in einem zusammenhängenden Speicherbereich bereitgestellt worden sein, wird der Wert *–1* zurückgeliefert.

Für nähere Informationen zur Verwendung der hier aufgeführten Methoden verweisen wir auf die Abschnitte 4.2.6 und 4.2.8.1.

Bereitstellung von Möglichkeiten zum lokalen Arbeiten

Zum lokalen Verarbeiten der Sequenzausschnitte werden dem Anwendungsprogramm Sequenzausschnittsmethoden zur Verfügung gestellt, die an die im Abschnitt 2.8 vorgestellte XML-Programmierschnittstelle DOM angelehnt sind. Diese Sequenzausschnittsmethoden erlauben sowohl ein lesendes als auch ein änderndes Arbeiten auf den lokal vorgehaltenen Sequenzausschnitten.

Für eine (exemplarische) Beschreibung der „DOM-artigen“ Sequenzausschnittsmethoden verweisen wir auf Abschnitt 6.5.1.1. Die Verwendung dieser Sequenzausschnittsmethoden wird zudem im Abschnitt 4.2.7 anhand eines konkreten Beispiels demonstriert. Weitere Details bezüglich der angebotenen Sequenzausschnittsmethoden finden sich in [Ott07].

Protokollierung der lokal vorgenommenen Änderungen

Auf die lokal vorgehaltenen Sequenzausschnitte darf vom Anwendungsprogramm *ausschließlich* mit Hilfe von Sequenzausschnittsmethoden zugegriffen werden. Zu den Aufgaben dieser Sequenzausschnittsmethoden gehört es, die lokal an den Sequenzausschnitten vorgenommenen Änderungen zu protokollieren. Im Rahmen unserer prototypischen Umsetzung wird diese Änderungsprotokollierung mit Hilfe von Markierungen realisiert (vgl. Abschnitt 6.5.3).

Jeder im Sequenzausschnitt enthaltene Knoten wird lokal als separates Objekt repräsentiert (Abschnitt 8.2.3). Ein solches Objekt enthält — zusätzlich zu den Knoten-bezogenen Informationen und den Zeigern auf andere Objekte — auch eine Statusinformation [Ott07]. Aus dieser Statusinformation geht hervor, ob der entsprechende Knoten (im Verlauf der lokalen Verarbeitung des Sequenzausschnitts) neu eingefügt, geändert oder gelöscht wurde. Es wird zwischen den folgenden vier Statusmöglichkeiten unterschieden:

'N' — (no change) Der Knoten wurde (im Verlauf der lokalen Verarbeitung) weder neu eingefügt noch geändert noch gelöscht.

'I' — (inserted) Der Knoten wurde neu eingefügt.

'U' — (updated) Der Knoten wurde geändert.

'D' — (deleted) Der Knoten wurde gelöscht.

Beim Erzeugen der lokalen Repräsentation des Sequenzausschnitts erhält jedes Objekt zunächst die Statusmarkierung *'N'*. Wird der mit einem Objekt korrespondierende Knoten im Verlauf der lokalen Verarbeitung geändert oder gelöscht, so wird die Statusinformation des entsprechenden Objekts mit *'U'* bzw. *'D'* überschrieben. Für die Aktualisierung der Statusinformation ist dabei diejenige Sequenzausschnittsmethode verantwortlich, mit deren Hilfe das Ändern bzw. Löschen vorgenommen wird.

Wird ein Knoten, der lokal neu eingefügt wurde, wieder gelöscht, so wird das ihn repräsentierende Objekt ebenfalls gelöscht. Über die „zwischenzeitliche Existenz" des entsprechenden Knotens werden also keinerlei (Änderungs-)Informationen vorgehalten. Wird hingegen ein Knoten gelöscht, bei dem es sich *nicht* um einen lokal neu eingefügten Knoten handelt, so wird das den Knoten repräsentierende Objekt *nicht* gelöscht, sondern lediglich als „gelöscht" markiert [Ott07].

Wird ein Knoten geändert, der zuvor neu eingefügt wurde, so wird die Statusinformation des mit diesem Knoten korrespondierenden Objekts *nicht* von *'I'* auf *'U'* aktualisiert. Im Rahmen der lokalen Änderungsprotokollierung wird der Knoten also so behandelt, als hätte er bereits beim Einfügen die geänderten Werte besessen.

Abbildung 8.11 fasst die Übergänge zwischen den Statusmöglichkeiten nochmals kompakt zusammen.

bisheriger Status	Status nach Änderung des Knotens	Status nach Löschen des Knotens
'N'	*'U'*	*'D'*
'I'	*'I'*	(Objekt wird gelöscht)
'U'	*'U'*	*'D'*
'D'	(nicht möglich)	(nicht möglich)

Abbildung 8.11: Übergänge zwischen den Statusmöglichkeiten

Bereitstellung der Sequenzausschnitts-bezogenen Änderungsinformationen

Beim Aufruf der Methode *provide_change_information()* wird eine (von der Sequenzcursorverwaltung verarbeitbare) Folge von Änderungseinträgen erzeugt, bei der jeder Änderungseintrag mit einem lokal neu eingefügten, lokal geänderten bzw. lokal gelöschten Knoten korrespondiert. Diese Folge von Änderungseinträgen wird in einem zusammenhängenden lokalen Speicherbereich abgelegt, aus dem sie dann von der Sequenzcursorverwaltung ausgelesen wird.

Zur Erzeugung der Folge von Änderungseinträgen werden sämtliche Objekte der lokalen Sequenzausschnittsrepräsentation durchlaufen [Ott07]. Dabei wird für jedes Objekt, das eine von *'N'* abweichende Statusinformation besitzt, ein geeigneter Änderungseintrag generiert. Ein solcher Änderungseintrag enthält sowohl die (Sequenzausschnitts-bezogene) Positionsnummer des von der Änderung betroffenen Knotens als auch Informationen zur genauen Art der entsprechenden Änderung. Für nähere Details verweisen wir auf [Ott07].

8.3 Fazit

Mit der Bereitstellung eines funktionsfähigen Prototyps [Kie06, Jat07, Böh07, Ott07] ist es uns gelungen, die Realisierbarkeit der wichtigsten Konzepte der Sequenzcursor-basierten Verarbeitung nachzuweisen. Mit dem Prototyp steht zudem eine Testplattform zur Verfügung, mit deren Hilfe sich die Verwendung der von uns vorgeschlagenen Sequenzcursoranweisungen praktisch erproben lässt. Darüber hinaus ermöglicht es der Prototyp, praktische Erfahrungen im Umgang mit dem SQL/XML:2006-Basisdatentyp `XML` zu sammeln.

Bei der Umsetzung der im Abschnitt 8.1.4 vorgestellten 4-Schichten-Architektur haben wir uns vor allem auf rein funktionale Aspekte konzentriert. Das (in der Praxis sehr wichtige) Ziel, eine gute Performance zu erreichen, spielte hingegen nur eine untergeordnete Rolle. Prototyp-bezogene Performancemessungen wären im Übrigen ohnehin nicht sinnvoll: Zum einen existiert noch kein „XQuery-Sequenz-fähiges“ DBMS-Produkt, welches für einen Performancevergleich herangezogen werden könnte, zum anderen wären etwaige Performanceergebnisse aufgrund der Verwendung der Sequenzsimulationskomponente sowieso nicht aussagekräftig.

Bei der bisherigen Nutzung des Prototyps wurden wichtige Erkenntnisse im Hinblick auf eine anwenderfreundliche Festlegung adäquater Sequenzausschnittsmethoden gewonnen. Dabei zeigte sich insbesondere, dass „DOM-artige“ Sequenzausschnittsmethoden gut für eine komfortable Verarbeitung der Sequenzausschnitte geeignet sind. Erfahrungen, die beim weiteren Einsatz des Prototyps gesammelt werden, sollen dazu beitragen, sowohl die Sequenzcursoranweisungen als auch die Sequenzausschnittsmethoden gezielt weiterzuentwickeln.

Kapitel 9

Zusammenfassung und Ausblick

Im Anschluss an eine kompakte Zusammenfassung der vorliegenden Arbeit im Abschnitt 9.1 gibt Abschnitt 9.2 einen Ausblick auf mögliche weiterführende Arbeiten.

9.1 Zusammenfassung der vorliegenden Arbeit

Im Rahmen der vorliegenden Arbeit wurden — nach einer Erläuterung grundlegender SQL- bzw. XML-bezogener Begriffe und Konzepte — die im Folgenden aufgeführten Themen behandelt.

XML-Unterstützung durch die SQL-Norm

In der Praxis besteht oftmals der Wunsch, XML-Daten und „traditionelle“ SQL-Daten *gemeinsam* verwalten und *integriert* auswerten zu können. Die SQL-Normungsgremien haben auf diesen Wunsch reagiert und SQL um XML-Funktionalität erweitert. Der hierzu im Jahr 2003 eigens neu eingeführte Normteil SQL/XML:2003 wurde bereits drei Jahre später durch eine grundlegend überarbeitete Fassung namens SQL/XML:2006 ersetzt. SQL/XML:2006 ist seitdem offizieller Bestandteil der (derzeit gültigen) SQL-Norm SQL:2003.

Wir sind darauf eingegangen, inwieweit XML-Werte (also XML-Dokumente bzw. XQuery-Sequenzen) von SQL/XML:2003 und SQL/XML:2006 unterstützt werden. Dabei haben wir hervorgehoben, dass in SQL/XML:2006 *beliebige* XQuery-Sequenzen als XML-Werte zulässig sind. Bei einem XML-Wert gemäß SQL/XML:2006 kann es sich also insbesondere auch um eine aus *mehreren* Sequenzeinträgen bestehende Sequenz handeln, deren Knoten mit Typinformationen angereichert sind.

XML-Unterstützung heutiger relationaler Datenbankprodukte

Der Wunsch nach einer SQL-basierten integrierten Verarbeitung von XML-Daten und „herkömmlichen“ SQL-Daten wurde (außer von den SQL-Normungsgremien) auch von den Herstellern relationaler DBMS-Produkte erkannt. Die drei Marktführer auf diesem Gebiet (Oracle, Microsoft und IBM) haben ihre Datenbankprodukte bereits mit beachtlicher XML-Funktionalität ausgestattet. Wir sind überzeugt, dass die XML-Unterstützung

in den künftigen Versionen der entsprechenden Produkte noch weiter ausgebaut werden wird. Wesentliche Entwicklungsarbeiten hierzu laufen derzeit u. a. bei IBM.

Wir haben untersucht, inwieweit Oracle 11g Release 1, MS SQL Server 2005 und IBM DB2 Universal Database 9.5 die Vorgaben von SQL/XML:2006 erfüllen. Dabei haben wir festgestellt, dass alle drei Produkte (trotz ihrer ansonsten beträchtlichen XML-Fähigkeiten) eine wesentliche Einschränkung aufweisen: Während gemäß SQL/XML:2006 als XML-Werte ausdrücklich XQuery-Sequenzen mit *mehreren* Sequenzeinträgen zulässig sind (also geordnete Folgen, die sich aus *mehreren* Bäumen und/oder atomaren Werten zusammensetzen), beschränken sich die Produkte auf XML-Dokumente (also einzelne Bäume).

Beispielszenario Kundenkartenverwaltung

Da bisher noch kein relationales Datenbankprodukt verfügbar ist, das XML-Werte gemäß SQL/XML:2006 unterstützt, existiert auch noch kein „echtes“, in der Praxis erprobtes SQL/XML:2006-Anwendungsszenario. Aus diesem Grund haben wir ein eigenes, realitätsnahes Beispielszenario eingeführt, das auf SQL/XML:2006 abgestimmt ist. Auf dieses Beispielszenario haben wir im Verlauf der Arbeit dann mehrfach Bezug genommen.

Im Fokus des Beispielszenarios steht ein Unternehmen, welches Kundenkarten ausgibt und verwaltet. Ein Kunde kann seine Kundenkarte z. B. bei einem Einkauf oder bei einer Autoanmietung einsetzen, um so genannte Bonusangebote zu erwerben. Die von einem Kunden in einer bestimmten Kategorie erworbenen Bonusangebote werden als XQuery-Sequenz in einer SQL/XML:2006-Tabelle verwaltet, wobei jeder Eintrag der Sequenz jeweils ein Bonusangebot repräsentiert.

Die Besonderheit unseres Beispielszenarios besteht darin, dass für die Verwaltung der Bonusdaten eine Tabelle genutzt wird, in der (außer Nicht-XML-Werten, wie beispielsweise den Kundennummern und den Kategoriebezeichnungen) XQuery-Sequenzen abgelegt sind, die aus *mehreren* Sequenzeinträgen bestehen.

Sequenzcursor-basierte Verarbeitung

Entsprechend SQL/XML:2006 kann ein Anfrageergebnis XQuery-Sequenzen enthalten, die sich aus *mehreren* Sequenzeinträgen zusammensetzen. Damit stellt sich unweigerlich die Frage, wie solche (in einem SQL-Anfrageergebnis enthaltenen) XQuery-Sequenzen von einem Anwendungsprogramm verarbeitbar sind. Eine befriedigende Antwort hierauf liefert (bisher) weder die SQL-Norm noch eine andere Norm noch ein DBMS-Produkt noch eine existierende Forschungsarbeit. Um diese Lücke zu schließen, haben wir das Verfahren der Sequenzcursor-basierten Verarbeitung eingeführt.

Das Grundprinzip unseres Ansatzes besteht darin, mit Hilfe von Sequenzcursorn Ausschnitte der im aktuellen Ergebnistupel enthaltenen XQuery-Sequenzen zu definieren. Diese Sequenzausschnitte werden dann ins Anwendungsprogramm übertragen und stehen dort für eine lokale Verarbeitung zur Verfügung. Sofern im Verlauf der lokalen Verarbeitung Änderungen an den Sequenzausschnitten vorgenommen werden, kann das Anwendungsprogramm entscheiden, ob diese Änderungen in die Datenbank (wieder)eingebracht werden sollen oder nicht.

Der Ablauf der Sequenzcursor-basierten Verarbeitung wurde von uns ausführlich erläutert. Die Beschreibung der einzelnen Verarbeitungsschritte erfolgte dabei anhand unseres Bei-

spielszenarios. Wir haben mögliche Erweiterungen des Sequenzcursor-basierten Verarbeitungsablaufs vorgestellt und denkbare alternative Verarbeitungsansätze diskutiert. Außerdem haben wir den von uns konzipierten Sequenzcursor-basierten Verarbeitungsablauf gegen bereits existierende Ansätze abgegrenzt.

Typed-value-orientierte Repräsentation

Wir haben gezeigt, dass sich die traditionelle Sequenzrepräsentation nicht eignet, um als Basis der Sequenzcursor-basierten Verarbeitung genutzt zu werden. Mit der typed-value-orientierten Repräsentation haben wir dann eine neuartige Möglichkeit vorgestellt, XQuery-Sequenzen zu repräsentieren. Diese von uns konzipierte Repräsentationsvariante ist speziell auf die Bedürfnisse der Sequenzcursor-basierten Verarbeitung abgestimmt und dient als Grundlage für das Positionieren der Sequenzcursor, das Definieren der Sequenzausschnitte sowie das lokale Arbeiten auf den Sequenzausschnitten.

Die Besonderheit der typed-value-orientierten Repräsentation besteht darin, dass die im getypten Wert eines E- oder A-Knotens enthaltenen atomaren Werte (unter gewissen Voraussetzungen) als eigenständige V-Knoten repräsentiert werden. Damit wird erreicht, dass auf die entsprechenden atomaren Werte unkompliziert zugegriffen werden kann.

Wir haben ausführlich beschrieben, wie sich XQuery-Sequenzen in die typed-value-orientierte Repräsentation überführen lassen. Es wurde ausdrücklich darauf hingewiesen, dass bei einer solchen Überführung *keine* Erweiterung des Informationsgehalts vorgenommen wird: Informationen, die gemäß des XQuery-Datenmodells ohnehin Bestandteil einer traditionell repräsentierten XQuery-Sequenz sind, werden im Rahmen der typed-value-orientierten Repräsentation lediglich auf eine andere Art und Weise dargestellt.

Sequenzcursor

Die Sequenzcursor stellen eine von uns neu eingeführte Cursorart dar. Sie werden im Rahmen der Sequenzcursor-basierten Verarbeitung verwendet, um die ins Anwendungsprogramm zu übertragenden Sequenzausschnitte festzulegen. Die Sequenzcursor werden hierfür zunächst geeignet innerhalb der (im aktuellen Ergebnistupel enthaltenen) XQuery-Sequenzen positioniert. Beim anschließenden Definieren der Sequenzausschnitte wird dann auf die positionierten Sequenzcursor Bezug genommen.

Die grundlegenden Eigenschaften der Sequenzcursor wurden von uns ausführlich erläutert. Desweiteren haben wir detailliert beschrieben, welche Möglichkeiten es für die Positionierung von Sequenzcursorn gibt. Hervorzuheben ist hierbei, dass bei einer Sequenzcursorpositionierung die Knotenart, der Knotentyp bzw. der Knotenname berücksichtigt werden kann.

Sequenzausschnitte

Entsprechend des Sequenzcursor-basierten Verarbeitungsmodells wird eine im aktuellen Ergebnistupel enthaltene (möglicherweise sehr große) XQuery-Sequenz i. d. R. nicht komplett ins Anwendungsprogramm übertragen. Es erfolgt stattdessen lediglich die Übertragung eines zuvor explizit definierten Ausschnitts der Sequenz. Dieser Sequenzausschnitt steht im Anwendungsprogramm dann für eine lokale Verarbeitung zur Verfügung.

Wir sind ausführlich auf die grundlegenden Eigenschaften der Sequenzausschnitte eingegangen. Dabei haben wir hervorgehoben, dass ein Sequenzausschnitt nicht zusammenhängend sein muss, sondern aus *mehreren* (nicht miteinander verbundenen) Sequenzausschnittsteilen bestehen kann. Bezogen auf unser Beispielszenario ist es damit z. B. möglich, einen Sequenzausschnitt zu definieren, der ausschließlich das erste und das letzte (von einem bestimmten Kunden in einer bestimmten Kategorie erworbene) Bonusangebot umfasst.

Vorschlag zur Erweiterung von SQL

Wir haben erläutert, um welche Sprachkonstrukte die Datenbanksprache SQL zu erweitern ist, damit eine SQL-seitige Unterstützung des Sequenzcursor-basierten Verarbeitungsablaufs gewährleistet wird. Bei diesen Sprachkonstrukten handelt es sich um die zehn von uns vorgeschlagenen Sequenzcursoranweisungen sowie um die beiden von uns konzipierten (zur Kommunikation zwischen Anwendungsprogramm und SQL-Laufzeitsystem dienenden) Datenstrukturen SQLSCFA und SQLSPCA. Die entsprechend erweiterte Fassung von SQL haben wir als SQL^+ bezeichnet.

Wir sind detailliert auf die Syntax und die Semantik der Sequenzcursoranweisungen sowie auf den Aufbau und die Nutzung der beiden Datenstrukturen eingegangen. Die konkrete Verwendung der Sequenzcursoranweisungen wurde zudem anhand unseres Beispielszenarios veranschaulicht.

Prototypische Realisierung

Durch die Bereitstellung eines funktionierenden Prototyps — in dessen Vorbereitung und Entwicklung mehrere Personenjahre Arbeit (insbesondere in Form von Diplomarbeiten) einflossen — konnten wir die Realisierbarkeit der wesentlichen Konzepte des Sequenzcursor-basierten Verarbeitungsmodells zeigen. Der Implementierung unseres Prototyps liegt eine 4-Schichten-Architektur zugrunde. Die einzelnen Schichten dieser Architektur wurden von uns detailliert beschrieben.

Der Prototyp basiert auf der nicht-erweiterten Variante des Sequenzcursor-basierten Verarbeitungsablaufs. Er kann (anstelle beliebiger SQL^+-Anweisungen) lediglich Sequenzcursoranweisungen verarbeiten und ist nicht für Performance-Messungen geeignet.

Der Prototyp kann genutzt werden, um die Verwendung der Sequenzcursoranweisungen praktisch zu erproben. Darüber hinaus lassen sich mit seiner Hilfe praktische Erfahrungen im Umgang mit dem SQL/XML:2006-Basisdatentyp `XML` sammeln.

9.2 Ausblick auf weiterführende Arbeiten

Für weiterführende Arbeiten ergeben sich (u. a.) die im Folgenden betrachteten Ansatzpunkte.

Weiterentwicklung des Prototyps

Der Prototyp sollte dahingehend weiterentwickelt werden, dass er auch die erweiterte Variante des Sequenzcursor-basierten Verarbeitungsablaufs unterstützt. Ein anderes Ziel beim

Ausbau des Prototyps sollte darin bestehen, den Prototyp in die Lage zu versetzen, beliebige SQL$^+$-Anweisungen verarbeiten zu können. Mit Hilfe des Prototyps ließe sich dann das Zusammenspiel zwischen den von uns eingeführten Sequenzcursoranweisungen und den „traditionellen“ SQL-Anweisungen praktisch erproben.

Bei einer Überarbeitung des Prototyps sollten (anders als bisher) auch Performance-Aspekte eine wichtige Rolle spielen. Dies hätte den Vorteil, dass der Prototyp künftig auch im Rahmen von Performance-Untersuchungen einsetzbar wäre.

Weiterentwicklung des Sprachvorschlags

Basierend auf Erkenntnissen, die beim Einsatz des Prototyps gewonnen werden, sollte eine gezielte Weiterentwicklung des (SQL$^+$-)Sprachvorschlags erfolgen.

Mit Hilfe des Prototyps sollte insbesondere untersucht werden, ob die von uns für eine Sequenzcursorpositionierung vorgesehenen Möglichkeiten „im praktischen Einsatz“ tatsächlich ausreichen oder ob zusätzliche Positionierungsvarianten wünschenswert sind. Eine entprechende Untersuchung sollte auch bezüglich der Möglichkeiten durchgeführt werden, die für eine Sequenzausschnittsdefinition zur Verfügung stehen.

Betrachtung weiterer Anwendungsszenarien

Es sollten weitere realitätsnahe Anwendungsszenarien entwickelt werden, die auf einer SQL-basierten integrierten Verarbeitung von XML-Daten und „herkömmlichen“ SQL-Daten beruhen und dabei XML-Werte gemäß SQL/XML:2006 voraussetzen. Diese Anwendungsszenarien könnten dann systematisch analysiert werden, um zusätzliche (bisher nicht bekannte) Anforderungen an die Sequenzcursor-basierte Verarbeitung zu identifizieren. Auf der Grundlage der so ermittelten Anforderungen sollte eine zielgerichtete Weiterentwicklung des Sequenzcursor-basierten Verarbeitungsmodells erfolgen.

Sobald in der Praxis die ersten *realen* SQL/XML:2006-Anwendungsszenarien existieren, sollten diese Anwendungsszenarien natürlich in die Betrachtungen einbezogen werden.

Stärkere Formalisierung des Sprachvorschlags

Idealerweise sollten die von uns vorgeschlagenen Sequenzcursoranweisungen Eingang in die offizielle SQL-Norm finden. Gleiches gilt für die von uns konzipierten (zur Kommunikation zwischen Anwendungsprogramm und SQL-Laufzeitsystem dienenden) Datenstrukturen SQLSCFA und SQLSPCA. Damit dieses Ziel erreicht werden kann, muss zunächst ein formaler SQL-Änderungsvorschlag erarbeitet werden, der dann bei den zuständigen SQL-Normungsgremien (DIN bzw. ISO) einzureichen ist. Dass dieser Änderungsvorschlag den Normungsprozess tatsächlich *erfolgreich* durchlaufen wird, ist im Vorfeld (natürlich) nicht garantiert.

In Vorbereitung des einzureichenden SQL-Änderungsvorschlags muss unser Sprachvorschlag stärker formalisiert und weiter präzisiert werden. Außerdem ist es erforderlich, unseren Sprachvorschlag mit anderen aktuellen Normentwicklungen abzustimmen.

Literaturverzeichnis

[BEA03] BEA Systems. *Streaming API For XML: JSR-173 Specification.* San Jose, Oktober 2003.

[BLFM05] T. Berners-Lee, R. Fielding und L. Masinter. *Uniform Resource Identifier (URI): Generic Syntax*, Januar 2005. Network Working Group, RFC 3986.

[Böh06] R. Böhme. Erproben der XML-Funktionalität der IBM-DB2-Alpha-Version und Erstellen geeigneter Testszenarien. Studienarbeit, Institut für Informatik, Friedrich-Schiller-Universität Jena, Januar 2006.

[Böh07] R. Böhme. Konzeption und Implementierung von Komponenten zur datenbankbasierten Verarbeitung von XML-Werten. Diplomarbeit, Institut für Informatik, Friedrich-Schiller-Universität Jena, März 2007.

[Bre07] U. Breymann. *C++. Einführung und professionelle Programmierung.* Hanser Fachbuch, München, 9. Auflage, April 2007.

[Bro04] M. Brosemann. Vergleich und Bewertung der XML-Funktionalität heutiger Datenbanksysteme. Studienarbeit, Institut für Informatik, Friedrich-Schiller-Universität Jena, Oktober 2004.

[Cha04] D. Chamberlin. *No need for namespace nodes.* World Wide Web Consortium (W3C), Februar 2004. Change proposal. [DM] IBM-DM-031.

[Cod70] E. F. Codd. A Relational Model of Data for Large Shared Data Banks. *Communications of the ACM*, 13(6):377–387, Juni 1970.

[Con96] The Unicode Consortium. *The Unicode Standard, Version 2.0.* Addison-Wesley, Reading, MA, August 1996.

[Con00] The Unicode Consortium. *The Unicode Standard Version 3.0.* Addison-Wesley, Reading, MA, Februar 2000.

[Con03] The Unicode Consortium. *The Unicode Standard 4.0.* Addison-Wesley, Reading, MA, August 2003.

[Con06] The Unicode Consortium. *The Unicode Standard 5.0.* Addison-Wesley, Reading, MA, November 2006.

[DD97] C. J. Date und H. Darwen. *A Guide to the SQL standard.* Addison-Wesley, Reading, MA, 4. Auflage, April 1997.

[EE03] R. Eckstein und S. Eckstein. *XML und Datenmodellierung*. dpunkt-Verlag, Heidelberg, 1. Auflage, November 2003.

[EM01] A. Eisenberg und J. Melton. SQL/XML and the SQLX Informal Group of Companies. *SIGMOD Record*, 30(3):105–108, September 2001.

[EM02] A. Eisenberg und J. Melton. SQL/XML is Making Good Progress. *SIGMOD Record*, 31(2):101–108, Juni 2002.

[EM04a] A. Eisenberg und J. Melton. Advancements in SQL/XML. *SIGMOD Record*, 33(3):79–86, September 2004.

[EM04b] A. Eisenberg und J. Melton. An Early Look at XQuery API for Java (XQJ). *SIGMOD Record*, 33(2):105–111, Juni 2004.

[EMK+04] A. Eisenberg, J. Melton, K. Kulkarni, J.-E. Michels und F. Zemke. SQL:2003 has been published. *SIGMOD Record*, 33(1):119–126, März 2004.

[Gei95] K. Geiger. *Inside ODBC: Developer's Guide to the Industry Standard for Database Connectivity*. Microsoft Press, Redmond, WA, August 1995.

[Gol05] C. Gollmick. Konzept, Realisierung und Anwendung nutzerdefinierter Replikation in mobilen Datenbanksystemen. Dissertation, Fakultät für Mathematik und Informatik, Friedrich-Schiller-Universität Jena, November 2005.

[HM05] E. R. Harold und W. S. Means. *XML in a nutshell*. O'Reilly Verlag, Köln, 3. Auflage, Januar 2005. Deutsche Ausgabe der 3. Auflage.

[HRS+05] M. Harren, M. Raghavachari, O. Shmueli, M. G. Burke, R. Bordawekar, I. Pechtchanski und V. Sarkar. XJ: Facilitating XML Processing in Java. In *Proceedings of The 14th International World Wide Web Conference 2005 (WWW2005)*, Seiten 278–287, Chiba, Mai 2005.

[HS00] A. Heuer und G. Saake. *Datenbanken: Konzepte und Sprachen*. MITP–Verlag, Bonn, 2. Auflage, 2000.

[HW04] F. Havekost und W. Willems. Siegeszug der Kundenkarte. *Geldidee*, (17):20–25, August 2004.

[IBM04a] IBM Corporation. *IBM DB2 Universal Database — SQL Reference Volume 1 (Version 8.2)*, 2004.

[IBM04b] IBM Corporation. *IBM DB2 Universal Database — XML Extender Administration and Programming (Version 8.2)*, 2004.

[IBM05] IBM Corporation. *DB2 Information Management Software — The IBM approach to unified XML/relational databases*, März 2005.

[IBM07a] IBM Corporation. *DB2 Version 9.5 for Linux, UNIX, and Windows — pureXML Guide*, 2007.

[IBM07b] IBM Corporation. *DB2 Version 9.5 for Linux, UNIX, and Windows — SQL Reference Volume 1*, 2007.

[IBM07c] IBM Corporation. *DB2 Version 9.5 for Linux, UNIX, and Windows — SQL Reference Volume 2*, 2007.

[ISO86] International Organization for Standardization (ISO), Genf. *Information processing – Text and Office Systems – Standard Generalized Markup Language (SGML)*, Oktober 1986. International Standard. ISO 8879:1986.

[ISO87] International Organization for Standardization (ISO), Genf. *Information technology – Database languages – SQL*, 1987. International Standard. ISO/IEC 9075:1987.

[ISO89] International Organization for Standardization (ISO), Genf. *Information technology – Database languages – SQL*, 1989. International Standard. ISO/IEC 9075:1989.

[ISO92] International Organization for Standardization (ISO), Genf. *Information technology – Database languages – SQL*, 1992. International Standard. ISO/IEC 9075:1992.

[ISO99] International Organization for Standardization (ISO), Genf. *Information technology – Database languages – SQL*, 1999. International Standard. ISO/IEC 9075:1999.

[ISO03a] International Organization for Standardization (ISO), Genf. *Information technology – Database languages – SQL*, Dezember 2003. International Standard. ISO/IEC 9075:2003.

[ISO03b] International Organization for Standardization (ISO), Genf. *Information technology – Database languages – SQL – Part 14: XML–Related Specifications (SQL/XML)*, Dezember 2003. International Standard. ISO/IEC 9075-14:2003.

[ISO03c] International Organization for Standardization (ISO), Genf. *Information technology – Database languages – SQL – Part 2: Foundation (SQL/Foundation)*, Dezember 2003. International Standard. ISO/IEC 9075-2:2003.

[ISO03d] International Organization for Standardization (ISO), Genf. *Information technology – Database languages – SQL – Part 3: Call-Level Interface (SQL/CLI)*, Dezember 2003. International Standard. ISO/IEC 9075-3:2003.

[ISO06] International Organization for Standardization (ISO), Genf. *Information technology – Database languages – SQL – Part 14: XML–Related Specifications (SQL/XML)*, November 2006. International Standard. ISO/IEC 9075-14:2006.

[ISO07] International Organization for Standardization (ISO), Genf. *Information technology – Database languages – SQL – Part 14: XML–Related Specifications (SQL/XML) – Technical Corrigendum 1*, April 2007. ISO/IEC 9075-14:2006/Cor.1:2007.

[Jat07] M. Jatho. Konzeption und Implementierung von Komponenten zur datenbankorientierten Verarbeitung von XML-Werten. Diplomarbeit, Institut für Informatik, Friedrich-Schiller-Universität Jena, März 2007.

[KBL05] M. Kifer, A. Bernstein und P. M. Lewis. *Database Systems: An Application-Oriented Approach*. Addison-Wesley, Boston, 2. Auflage, 2005.

[Kie05] C. Kiewitt. XQuery-Datenmodell und Repräsentation von XML-Werten in C++. Studienarbeit, Institut für Informatik, Friedrich-Schiller-Universität Jena, Juli 2005.

[Kie06] C. Kiewitt. Konzeption und Implementierung von Komponenten zur datenbankorientierten Verarbeitung von XML-Werten. Diplomarbeit, Institut für Informatik, Friedrich-Schiller-Universität Jena, Juli 2006.

[KL03] M. Kempa und V. Linnemann. Type Checking in XOBE. In *Tagungsband der 10. GI-Fachtagung Datenbanksysteme für Business, Technologie und Web (BTW), Leipzig*, Band P-26 aus *Lecture Notes in Informatics*, Seiten 227–246. Gesellschaft für Informatik, Februar 2003.

[KM02] M. Klettke und H. Meyer. *XML & Datenbanken: Konzepte, Sprachen und Systeme*. dpunkt-Verlag, Heidelberg, 1. Auflage, Dezember 2002.

[KM06] C. Kiewitt und T. Müller. SQL-basierte Datenbankzugriffe und XML: Eine 4-Schichten-Architektur. In *Tagungsband zum 18. GI-Workshop über Grundlagen von Datenbanken*, Seiten 85–89. Institut für Informatik, Martin-Luther-Universität Halle-Wittenberg. Wittenberg, Juni 2006.

[Kol06] S. Koll. Schlange mit Biss. *Möglich — Neue Perspektiven für Ihr Unternehmen*, (2):24–26, 2006.

[Kul03] K. Kulkarni. *Host Language mapping for XML type*. Santa Fe, Januar 2003. Change proposal. ISO/IEC JTC1/SC32 WG3:ZSH-083R2.

[Loy06] Loyalty Partner. *Aral wird Partner von PAYBACK*, April 2006. Pressemitteilung vom 07.04.2006.

[LS04a] W. Lehner und H. Schöning. XQuery — ein Überblick. *Datenbank-Spektrum*, (11):23–32, November 2004.

[LS04b] W. Lehner und H. Schöning. *XQuery — Grundlagen und fortgeschrittene Methoden*. dpunkt-Verlag, Heidelberg, 1. Auflage, August 2004.

[Mar05] R. Marhold. Konzeption eines SQL/XML:2007-Anwendungsszenarios. Diplomarbeit, Institut für Informatik, Friedrich-Schiller-Universität Jena, September 2005.

[MB02] W. S. Means und M. A. Bodie. *The Book of SAX — The simple API for XML*. No Starch Press, San Francisco, Juni 2002.

[MB06] J. Melton und S. Buxton. *Querying XML: XQuery, XPath, and SQL/XML in Context*. Elsevier, Amsterdam, 1. Auflage, März 2006.

[ME00] J. Melton und A. Eisenberg. *Understanding SQL and Java Together: A Guide to SQLJ, JDBC, and Related Technologies*. Morgan Kaufmann, San Francisco, CA, 1. Auflage, Mai 2000.

[Mel07] J. Melton. *XQuery API for Java (XQJ) 1.0 Specification: JSR-225*, Mai 2007. Public Draft Specification.

[Mic07a] Microsoft. *Guidelines and Limitations in Using XML Schema Collections on the Server*, September 2007. SQL Server 2005 Books Online.

[Mic07b] Microsoft. *XML Data Type*, September 2007. SQL Server 2005 Books Online.

[Mic07c] Microsoft. *XQuery Against the XML Data Type*, September 2007. SQL Server 2005 Books Online.

[MKF+03] J.-E. Michels, K. Kulkarni, C. Farrar, A. Eisenberg, N. Mattos und H. Darwen. The SQL Standard. In *it – Information Technology*, 1/2003, Seiten 30–38, Februar 2003.

[MO08] T. Müller und S. Ott. Sequenzcursorkonzept zur Verarbeitung von XQuery-Sequenzen in SQL/XML:2006-Anfrageergebnissen: Konzepte und prototypische Umsetzung. *Informatik Forschung und Entwicklung*, 2008. Wird zur Veröffentlichung eingereicht.

[MR05] T. Müller und G. Rabinovitch. Das Navigationsbasis-Modell zur Unterstützung der cursorbasierten Übergabe von XQuery-Sequenz-Ausschnitten zwischen Datenbanksystem und Anwendungsprogramm. In *Berliner XML-Tage 2005 — Tagungsband*, Seiten 135–146. Humboldt-Universität zu Berlin und Freie Universität Berlin, September 2005.

[Mül03] T. Müller. Architektur und Realisierung eines Replication Proxy Server zur Unterstützung neuartiger mobiler Datenbankanwendungen. Diplomarbeit, Institut für Informatik, Friedrich-Schiller-Universität Jena, April 2003.

[Mül04] T. Müller. SQL-basierte Datenbankzugriffe und XML: Klassifizierung von Anwendungsprogrammen. In *Tagungsband zum 16. GI-Workshop über Grundlagen von Datenbanken*, Seiten 88–92. Institut für Informatik, Heinrich-Heine-Universität Düsseldorf. Monheim am Rhein, Juni 2004.

[Mül05] T. Müller. SQL-basierte Datenbankzugriffe und XML: Verarbeitung von Anfrageergebnissen in Anwendungsprogrammen. In *Tagungsband zum 17. GI-Workshop über Grundlagen von Datenbanken*, Seiten 94–98. Institut für Informatik, Martin-Luther-Universität Halle-Wittenberg. Wörlitz, Mai 2005.

[Mül06] T. Müller. Sequenzcursor-basierte Verarbeitung von XML-Werten in SQL:2007-Anfrageergebnissen. In *XML-Tage 2006 in Berlin — Tagungsband*, Seiten 99–110. Humboldt-Universität zu Berlin und Freie Universität Berlin, September 2006.

[Neu92] K. Neumann. Kopplungsarten von Programmiersprachen und Datenbanksprachen. *Informatik-Spektrum*, 15(4):187–194, August 1992.

[Neu96] K. Neumann. *Datenbanktechnik für Anwender*. Hanser Fachbuch, München, 1. Auflage, 1996.

[NT05] M. Nist und S. Thorhold. Allgemeiner Überblick über XML 1.0. Studienarbeit, Institut für Informatik, Friedrich-Schiller-Universität Jena, Januar 2005.

[ÖCKM06] F. Özcan, D. Chamberlin, K. Kulkarni und J.-E. Michels. Integration of SQL and XQuery in IBM DB2. *IBM Systems Journal*, 45(2):245–270, April 2006.

[OP04] A. Osterhold und P. Pistor. Die SQL-Normen. *DIN Mitteilungen*, (4):27–36, April 2004. Übersetzte und aktualisierte Fassung des Artikels "The SQL Standard" von J.-E. Michels, K. Kulkarni, C. Farrar, A. Eisenberg, N. Mattos und H. Darwen.

[ORA07a] ORACLE Corporation. *Oracle Database — SQL Language Reference, 11g Release 1 (11.1)*, September 2007.

[ORA07b] ORACLE Corporation. *Oracle Database 11g XML DB Technical Overview — An Oracle White Paper*, Juli 2007.

[ORA07c] ORACLE Corporation. *Oracle XML DB — Developer's Guide, 11g Release 1 (11.1)*, September 2007.

[ORA07d] ORACLE Corporation. *Pro*C/C++ Programmer's Guide, 11g Release 1 (11.1)*, Juli 2007.

[Ott07] S. Ott. Konzeption und Implementierung von Komponenten zur datenbankbasierten Verarbeitung von XML-Werten. Diplomarbeit, Institut für Informatik, Friedrich-Schiller-Universität Jena, Juli 2007.

[Pet06] D. Petkovic. *SQL Server 2005: Eine umfassende Einführung.* dpunkt-Verlag, Heidelberg, 1. Auflage, März 2006.

[PN06] M. Päßler und M. Nicola. Native XML-Unterstützung in DB2 Viper. *Datenbank-Spektrum*, (17):42–47, Mai 2006.

[Rab05] G. Rabinovitch. Sequenzcursor-Konzept zur Übergabe von XML-Werten zwischen Datenbanksystem und Anwendungsprogramm. Diplomarbeit, Institut für Informatik, Friedrich-Schiller-Universität Jena, Juli 2005.

[Ree00] G. Reese. *Database Programming with JDBC and Java.* O'Reilly, Sebastopol, CA, 2. Auflage, September 2000.

[Rei05] J. Reichel. Programmierschnittstellen für XML. Studienarbeit, Institut für Informatik, Friedrich-Schiller-Universität Jena, Januar 2005.

[Rys05] M. Rys. XML and Relational Database Management Systems: Inside Microsoft SQL Server 2005. In *SIGMOD Conference*, Seiten 958–962, Juni 2005.

[Sch05] H. Schuhart. Persistenz- und Transaktionskonzepte in XOBE. In *Tagungsband zum 17. GI-Workshop über Grundlagen von Datenbanken*, Seiten 122–127. Institut für Informatik, Martin-Luther-Universität Halle-Wittenberg. Wörlitz, Mai 2005.

[SL05] H. Schuhart und V. Linnemann. Valid Updates for Persistent XML Objects. In *Tagungsband der 11. GI-Fachtagung Datenbanksysteme für Business, Technologie und Web (BTW), Karlsruhe*, Band P-65 aus *Lecture Notes in Informatics*, Seiten 245–264. Gesellschaft für Informatik, März 2005.

[Str00] B. Stroustrup. *Die C++ Programmiersprache.* Addison-Wesley, München, 4. Auflage, Mai 2000.

[Sun06] Sun Microsystems. *JDBC 4.0 Specification: JSR-221.* Santa Clara, November 2006. Final Release.

[Tür03] C. Türker. *SQL:1999 & SQL:2003: objektrelationales SQL, SQLJ & SQL/XML.* dpunkt-Verlag, Heidelberg, 1. Auflage, Februar 2003.

[Von05] H. Vonhoegen. *Einstieg in XML.* Galileo Computing, Bonn, 3. Auflage, März 2005.

[W3C98a] World Wide Web Consortium (W3C). *Document Object Model (DOM) Level 1 Specification (Version 1.0)*, Oktober 1998. W3C Recommendation.

[W3C98b] World Wide Web Consortium (W3C). *Extensible Markup Language (XML) 1.0*, Februar 1998. W3C Recommendation.

[W3C99a] World Wide Web Consortium (W3C). *Namespaces in XML*, Januar 1999. W3C Recommendation.

[W3C99b] World Wide Web Consortium (W3C). *XML Path Language (XPath) Version 1.0*, November 1999. W3C Recommendation.

[W3C99c] World Wide Web Consortium (W3C). *XSL Transformations (XSLT) Version 1.0*, November 1999. W3C Recommendation.

[W3C00] World Wide Web Consortium (W3C). *Extensible Markup Language (XML) 1.0 (Second Edition)*, Oktober 2000. W3C Recommendation.

[W3C01a] World Wide Web Consortium (W3C). *XML Base*, Juni 2001. W3C Recommendation.

[W3C01b] World Wide Web Consortium (W3C). *XML Information Set*, Oktober 2001. W3C Recommendation.

[W3C01c] World Wide Web Consortium (W3C). *XML Linking Language (XLink) Version 1.0*, Juni 2001. W3C Recommendation.

[W3C01d] World Wide Web Consortium (W3C). *XML Schema Part 0: Primer*, Mai 2001. W3C Recommendation.

[W3C03] World Wide Web Consortium (W3C). *XPointer Framework*, März 2003. W3C Recommendation.

[W3C04a] World Wide Web Consortium (W3C). *Document Object Model (DOM) Level 3 Core Specification (Version 1.0)*, April 2004. W3C Recommendation.

[W3C04b] World Wide Web Consortium (W3C). *Extensible Markup Language (XML) 1.0 (Third Edition)*, Februar 2004. W3C Recommendation.

[W3C04c] World Wide Web Consortium (W3C). *Extensible Markup Language (XML) 1.1*, Februar 2004. W3C Recommendation.

[W3C04d] World Wide Web Consortium (W3C). *XML Information Set (Second Edition)*, Februar 2004. W3C Recommendation.

[W3C04e] World Wide Web Consortium (W3C). *XML Schema Part 0: Primer Second Edition*, Oktober 2004. W3C Recommendation.

[W3C04f] World Wide Web Consortium (W3C). *XML Schema Part 1: Structures Second Edition*, Oktober 2004. W3C Recommendation.

[W3C04g] World Wide Web Consortium (W3C). *XML Schema Part 2: Datatypes Second Edition*, Oktober 2004. W3C Recommendation.

[W3C06a] World Wide Web Consortium (W3C). *Extensible Markup Language (XML) 1.0 (Fourth Edition)*, August 2006. W3C Recommendation.

[W3C06b] World Wide Web Consortium (W3C). *Extensible Markup Language (XML) 1.1 (Second Edition)*, August 2006. W3C Recommendation.

[W3C06c] World Wide Web Consortium (W3C). *Namespaces in XML 1.0 (Second Edition)*, August 2006. W3C Recommendation.

[W3C06d] World Wide Web Consortium (W3C). *XML Inclusions (XInclude) Version 1.0 (Second Edition)*, November 2006. W3C Recommendation.

[W3C07a] World Wide Web Consortium (W3C). *SOAP Version 1.2 Part 0: Primer (Second Edition)*, April 2007. W3C Recommendation.

[W3C07b] World Wide Web Consortium (W3C). *Web Services Description Language (WSDL) Version 2.0 Part 0: Primer*, Mai 2007. W3C Proposed Recommendation.

[W3C07c] World Wide Web Consortium (W3C). *XML Path Language (XPath) 2.0*, Januar 2007. W3C Recommendation.

[W3C07d] World Wide Web Consortium (W3C). *XQuery 1.0: An XML Query Language*, Januar 2007. W3C Recommendation.

[W3C07e] World Wide Web Consortium (W3C). *XQuery 1.0 and XPath 2.0 Data Model (XDM)*, Januar 2007. W3C Recommendation.

[W3C07f] World Wide Web Consortium (W3C). *XQuery Update Facility 1.0*, August 2007. W3C Working Draft.

[Wil07] A. Willemer. *Einstieg in C++*. Galileo Computing, Bonn, 3. Auflage, Mai 2007.

[Yan06] Q. Yan. Typed-value-orientierte Repräsentation von XQuery-Sequenzen. Studienarbeit, Institut für Informatik, Friedrich-Schiller-Universität Jena, November 2006.

[Zem04a] F. Zemke. *Moving to the XQuery data model*. Xian, April 2004. Change proposal. ISO/IEC JTC1/SC32 WG3:SIA-040.

[Zem04b] F. Zemke. *XMLQuery*. Xian, April 2004. Change proposal. ISO/IEC JTC1/SC32 WG3:SIA-042.

[ZRE+03] F. Zemke, B. Reinwald, A. Eisenberg, M. Rys, Z. Liu und A. Manikutty. *Moving to the XML Infoset data model*. Santa Fe, Januar 2003. Change proposal. ISO/IEC JTC1/SC32 WG3:ZSH-077R2.

[ZRK+04] F. Zemke, M. Rys, K. Kulkarni, J.-E. Michels, B. Reinwald, F. Ozcan, Z. Liu, I. Davis und K. Hare. *XMLTable*. Xian, Mai 2004. Change proposal. ISO/IEC JTC1/SC32 WG3:SIA-051.

Ehrenwörtliche Erklärung
zur Eröffnung des Promotionsverfahrens

Hiermit erkläre ich,

- dass mir die Promotionsordnung der Fakultät für Mathematik und Informatik der Friedrich-Schiller-Universität Jena bekannt ist,
- dass ich die Dissertation selbst angefertigt und alle von mir benutzten Hilfsmittel, persönlichen Mitteilungen und Quellen in meiner Arbeit angegeben habe,
- dass mich bei der Auswahl und Auswertung des Materials sowie bei der Herstellung des Manuskripts Herr Prof. Dr. Klaus Küspert (und sonst niemand) unterstützt hat,
- dass ich die Hilfe eines Promotionsberaters nicht in Anspruch genommen habe und dass Dritte weder unmittelbar noch mittelbar geldwerte Leistungen von mir für Arbeiten erhalten haben, die im Zusammenhang mit dem Inhalt der vorgelegten Dissertation stehen,
- dass ich die Dissertation noch nicht als Prüfungsarbeit für eine staatliche oder andere wissenschaftliche Prüfung eingereicht habe und
- dass ich die gleiche, eine in wesentlichen Teilen ähnliche oder eine andere Abhandlung nicht bei einer anderen Hochschule als Dissertation eingereicht habe.

Jena, 2. April 2008

Selbstständigkeitserklärung

Ich erkläre, dass ich die vorliegende Arbeit selbstständig und nur unter Verwendung der angegebenen Hilfsmittel und Literatur angefertigt habe.

Jena, 2. April 2008

Lebenslauf

Persönliche Daten

Name	Thomas Müller
Geburtsdatum	5. Juli 1976
Geburtsort	Sondershausen
Familienstand	verheiratet, 1 Kind

Schulische und universitäre Ausbildung

1983 – 1991	Polytechnische Oberschule „Hans Beimler“ in Greußen
1991 – 1995	Mathematisch-naturwissenschaftlicher Spezialschulteil des Albert-Schweitzer-Gymnasiums in Erfurt, Abschluss mit Abitur
1996 – 2003	Diplomstudium Informatik mit Nebenfach Psychologie an der Friedrich-Schiller-Universität Jena, Vertiefungsrichtung: Praktische Informatik, Schwerpunkt: Datenbanksysteme, Abschluss als Diplom-Informatiker
	Examenspreis des Rektors

Stipendien

1996	Aufnahme in die Studienstiftung des deutschen Volkes
2000 – 2001	Stipendiat der DaimlerChrysler AG

Zivildienst und beruflicher Werdegang

1995 – 1996	Zivildienst beim Deutschen Roten Kreuz in Sondershausen
2000 – 2001	Praktikum am IBM Silicon Valley Lab in San Jose, Kalifornien, USA
seit 2003	Wissenschaftlicher Mitarbeiter am Lehrstuhl für Datenbanken und Informationssysteme der Fakultät für Mathematik und Informatik der Friedrich-Schiller-Universität Jena

Jena, 2. April 2008